Vorlesung

Effiziente Algorithmen

SS 2010

Prof.Dr. Georg Schnitger

12. April 2010

Inhaltsverzeichnis

Kapitel 1

Einführung

Die folgenden Quellen ergänzen und vertiefen das Skript:

- Für das Kapitel über randomisierte Algorithmen die Textbücher [ML] und [MR].

- Für das Gebiet der on-line Algorithmen das Textbuch [BE]

Der erste Teil des Skripts behandelt den Entwurf und die Analyse randomisierter Algorithmen, wobei besonders Entwurfsmethoden und Random Walks betont werden. Markoff-Ketten helfen in der Durchführung von Stichproben und besitzen vielfältige Anwendungen, wie etwa Simulated Annealing, eine speicherplatz-effiziente Suche in ungerichteten Graphen, Pageranking in der Suchmaschine Google und die Volumenbestimmung konvexer Mengen.

On-line Algorithmen werden im zweiten Teil des Skipts behandelt. On-line Algorithmen spielen aufgrund der zunehmenden Interaktivität eine immer wesentlichere Rolle, und gerade hier finden sich wichtige Anwendungen randomisierter Algorithmen. Wir beginnen mit den für die Analyse wichtigen Grundlagen.

1.1 Wichtige Ungleichungen

Im Folgenden werden wir die Ungleichung von Cauchy-Schwartz benutzen.

Lemma 1.1 *Es seien x und y Vektoren im $\mathbb{R}^n$. Dann gilt*

$$(x, y) \leq ||x|| \cdot ||y||.$$

Beweis: Wir betrachten das innere Produkt (u, v) von zwei Vektoren u und v der Norm 1. Der Wert des inneren Produkts stimmt mit dem Kosinus des Winkels zwischen u und v überein und ist deshalb durch 1 nach oben beschränkt. Also folgt die Behauptung, wenn wir u durch $\frac{x}{||x||}$ und v durch $\frac{y}{||y||}$ ersetzen. □

Wir benötigen auch eine Abschätzung von $1 - x$ durch die e-Funktion.

Lemma 1.2 *Für jedes $x > -1$ gilt*

$$e^{x/(1+x)} \leq 1 + x \leq e^x.$$

Weiterhin gilt $1 + x \leq e^x$ für alle $x \in \mathbb{R}$.

Beweis: Wir betrachten zuerst den Fall $x \geq 0$. Der natürliche Logarithmus ist eine konkave Funktion, und deshalb ist die Steigung der Sekante durch die Punkte $(1,0)$ and $(1+x, \ln(1+x))$ nach unten durch die Steigung $1/(1+x)$ der Tangente in $(1+x, \ln(1+x))$ und nach oben durch die Steigung 1 der Tangente in $(1,0)$ beschränkt. Wir erhalten deshalb die Ungleichung

$$\frac{1}{1+x} \leq \frac{\ln(1+x) - \ln(1)}{(1+x)-1} = \frac{\ln(1+x)}{x} \leq 1 \tag{1.1}$$

oder äquivalent

$$\frac{x}{1+x} \leq \ln(1+x) \leq x. \tag{1.2}$$

Die erste Behauptung folgt also für $x \geq 0$ durch Exponentieren. Für $x \in]-1, 0]$ müssen wir nur beobachten, dass diesmal die Ungleichung

$$1 \leq \frac{\ln(1+x) - \ln(1)}{(1+x)-1} \leq \frac{1}{1+x} \tag{1.3}$$

gilt: Das Argument ist analog zum Beweis der Ungleichung (1.2). Wir erhalten wieder Ungleichung (1.2), wenn wir (1.3) mit der negativen Zahl x multiplizieren.

Aber damit erhalten wir auch $1+x \leq e^x$ für alle reellen Zahlen, da $1+x$ nicht-positiv für $x \leq -1$ ist und e^x immer positiv ist. □

Die nächste Ungleichung ist für die Abschätzung von Binomialkoeffizienten wichtig. Sie folgt aus der Stirling Formel

$$n! = \sqrt{2\pi \cdot n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \frac{1}{12 \cdot n} + O(\frac{1}{n^2})\right).$$

Lemma 1.3 *Es ist*

$$\binom{n}{k} \leq \left(\frac{n \cdot e}{k}\right)^k.$$

1.2 Grundlagen aus der Stochastik

Definition 1.4 Sei U eine endliche Menge.

(a) Eine Wahrscheinlichkeitsverteilung über U ist eine Funktion $\pi : U \to [0,1]$, so dass $\sum_{u \in U} \pi(u) = 1$.

(b) Die Elemente von U heißen Elementarereignisse, Teilmengen von U heißen Ereignisse. Die Wahrscheinlichkeit eines Ereignisses $V \subseteq U$ wird durch

$$\text{prob}[V] = \sum_{u \in V} \pi(u)$$

definiert.

(c) Es seien V_1 und V_2 zwei Ereignisse. Dann ist

$$\text{prob}[\ V_1 \mid V_2\ ] = \frac{\text{prob}[V_1 \cap V_2]}{\text{prob}[V_2]}$$

die bedingte Wahrscheinlichkeit von V_1 bezüglich V_2.

(d) Eine Zufallsvariable über U ist eine Funktion $X : U \to \mathbb{R}$. Der Erwartungswert von X wird durch

$$E[X] = \sum_{u \in U} \pi(u) \cdot X(u)$$

und die Varianz durch

$$V[X] = E[\ X^2 - E[X]^2\ ]$$

definiert.

Beispiel 1.1 Die **Gleichverteilung** oder **uniforme Verteilung** auf U weist jedem Elementarereignis $u \in U$ die Wahrscheinlichkeit $\frac{1}{|U|}$ zu.

Wir werfen eine Münze n-mal und definieren die Zufallsvariable X als die Anzahl der Versuche mit Ausgang Wappen. (X ist also über $U = \{\text{Wappen, Zahl}\}^n$ definiert.) Wenn p die Wahrscheinlichkeit für Wappen ist, dann ist

$$p_k = \text{prob}[\ X = k\ ] = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}.$$

$p = (p_0, \ldots, p_n)$ ist die **Binomialverteilung** mit den Parametern n und p. Beachte, dass $E[\ X\ ] = n \cdot p$ und $V[\ X\ ] = n \cdot p \cdot (1-p)$.

Wir werfen solange eine Münze, bis wir den Ausgang Wappen erhalten. Die Zufallsvariable X halte diese Anzahl der Versuche fest. (Die Zufallsvariable ist über $U = \{\ \text{Zahl}^n \cdot \text{Wappen} \mid n \in \mathbb{N}\}$ definiert. Diesmal sind also abzählbar unendlich viele Elementarereignisse gegeben.) Wenn p wiederum die Wahrscheinlichkeit für Wappen ist, dann ist

$$p_k = \text{prob}[\ X = k\ ] = (1-p)^{k-1} \cdot p.$$

$p = (p_0, \ldots)$ ist eine **geometrische Verteilung**. Beachte, dass

$$E[X] = \sum_{k=1}^{\infty} k \cdot (1-p)^{k-1} \cdot p = \frac{1}{p}.$$

Aufgabe 1

Uns steht eine stochastische Zufallsquelle zur Verfügung, die das Bit 0 (bzw. 1) mit Wahrscheinlichkeit $\frac{1}{2}$ ausgibt.

- Gegeben ist eine Zielverteilung $(p_1, \ldots, p_n)$. Entwirf einen Algorithmus, der mit Wahrscheinlichkeit p_i den Wert i ausgibt und dazu in der Erwartung nur $O(log(n))$ Bits aus der Zufallsquelle benötigt. Beachte, dass $p_1, \ldots, p_n$ beliebige reelle Zahlen sind.
- Bestimme die worst-case Zahl von Zufallsbits, mit der das obige Problem für $p_1 = p_2 = p_3 = \frac{1}{3}$ gelöst werden kann.

Aufgabe 2

Wir spielen ein Spiel gegen einen Gegner. Der Gegner denkt sich zwei Zahlen aus und schreibt sie für uns nicht sichtbar auf je einen Zettel. Wir wählen *zufällig* einen Zettel und lesen die darauf stehende Zahl. Sodann haben wir die Möglichkeit, diese Zahl zu behalten oder sie gegen die uns unbekannt gebliebene Zahl zu tauschen. Sei x die Zahl, die wir am Ende haben, und y die andere. Dann ist unser (möglicherweise negativer) Gewinn $x - y$.

- Wir betrachten Strategien S_t der Form *„Gib Zahlen $< t$ zurück und behalte diejenigen $\geq t$“*. Analysiere den Erwartungswert $E_{x,y}(Gewinn(S_t))$ des Gewinns dieser Strategie in Abhängigkeit von t, x und y.
- Gib eine randomisierte Strategie an, die für beliebige $x \neq y$ einen positiven erwarteten Gewinn für uns aufweist.

Beispiel 1.2 **Das Sekretär Problem**.

Wir möchten eine Stelle mit einem möglichst kompetenten Bewerber besetzen. Wir haben allerdings ein Problem: Wann immer sich ein Bewerber vorstellt, müssen wir am Ende des Bewerbungsgesprächs dem Bewerber absagen oder aber zusagen (und damit den Bewerbungsprozess beenden). Gibt es eine Strategie, die es uns erlaubt mit guter Wahrscheinlichkeit die beste Bewerbung auszuwählen?

Im allgemeinen Fall ist die Antwort sicherlich negativ: Wenn die erste Bewerbung bereits die beste ist, dann wird es uns schwerfallen sofort „zuzuschlagen“, da ja noch viele andere, möglicherweise bessere Bewerbungen anstehen. Erstaunlicherweise ist die Antwort aber positiv, wenn wir fordern, dass

- die Zahl n der Bewerber von vornherein bekannt ist und
- dass die Bewerber in zufälliger Reihenfolge zum Bewerbungsgespräch erscheinen.

n Bewerber $B_1, \ldots, B_n$ werden also eingeladen und ihre Reihenfolge π wird zufällig nach der Gleichverteilung ausgewürfelt. Nach dem Gespräch mit Bewerber B_i vergeben wir eine Note N_i. Wie können wir einen Bewerber mit höchster Note N_i auswählen, obwohl wir nach jedem Gespräch zu- oder absagen müssen? Wir betrachten die folgende Strategie für einen Parameter r mit $1 \leq r \leq n$:

(1) Wir sagen den ersten r Bewerbern ab, bestimmen aber die Höchstnote N, die von einem dieser Bewerber erreicht wird.

(2) Wir sagen dem ersten verbleibenden Bewerber zu, der mindestens mit N benotet wird. Wird die Höchstnote nicht mehr erreicht, dann müssen wir dem letzten Bewerber zusagen.

Wie gut ist diese Strategie und insbesondere, wie sollte r gewählt werden? Wir nehmen an, dass der beste Bewerber an Position opt erscheint. Wenn opt $\leq r$, dann haben wir verloren, denn wir haben dem besten Bewerber abgesagt.

Nehmen wir also an, dass opt $> r$ gilt, und der beste Bewerber bleibt also im Rennen. Beachte, dass wir genau dann erfolgreich sind, wenn der **zweitbeste unter den ersten opt Bewerbern** zu den ersten r Bewerbern gehört, also automatisch abgewiesen wird. Warum?

- Wenn der zweitbeste zu den ersten r Bewerbern gehört, der beste aber nicht, dann wird unsere Strategie den besten Bewerber auswählen.
- Wenn der zweitbeste nicht zu den ersten r Bewerbern gehört, dann wird unsere Strategie den Bewerbungsprozess vor dem besten Bewerber abbrechen, denn der zweitbeste erscheint ja vor Position opt.

Der zweitbeste kann an jeder der opt-1 Positionen erscheinen. Wenn also opt $> r$ gilt, dann erscheint der zweitbeste mit Wahrscheinlichkeit $r/(\text{opt} - 1)$ unter den ersten r Bewerbern und damit ist $r/(\text{opt}-1)$ die Erfolgswahrscheinlichkeit unserer Strategie. Der beste Bewerber erscheint mit Wahrscheinlichkeit $1/n$ auf irgendeiner fixierten Position und die Erfolgswahrscheinlichkeit p unserer Strategie ist

$$p = \sum_{\text{opt}=r+1}^{n} \frac{1}{n} \cdot \frac{r}{\text{opt} - 1}.$$

Wir approximieren die Summe durch das entsprechende Integral und erhalten

$$p = \sum_{\text{opt}=r+1}^{n} \frac{1}{n} \cdot \frac{r}{\text{opt}-1} \approx \frac{r}{n} \cdot \int_{x=r}^{n} \frac{1}{x}\, dx = \frac{r}{n} \cdot (\ln(n) - \ln(r)) = -\frac{r}{n} \cdot \ln(\frac{r}{n}).$$

Um die Erfolgswahrscheinlichkeit zu maximieren, müssen r so bestimmen, dass

$$f(r) = -\frac{r}{n} \cdot \ln(\frac{r}{n})$$

maximiert wird. Alternativ müssen wir das Maximum von $g(x) = -x \cdot \ln(x)$ über dem Intervall $[0, 1]$ maximieren. Wir differenzieren und erhalten

$$g'(x) = -\ln(x) - x \cdot \frac{1}{x} = -\ln(x) - 1.$$

Wenn wir $g'(x) = 0$ fordern, werden wir auf

$$\ln(x) = -1$$

und damit auf $x = 1/e$ geführt.

Satz 1.5 *Betrachte die Strategie, die die ersten $r = n/e$ Bewerber ablehnt und den ersten Bewerber akzeptiert, dessen Note mindestens so hoch ist wie die der ersten r Bewerber. Dann wird der beste Bewerber mit einer Wahrscheinlichkeit von ungefähr $\frac{1}{e}$ bestimmt.*

Als Nächstes beschreiben wir das „Monty Hall Problem“. In einer Game Show ist hinter einer von drei Türen ein Preis verborgen. Ein Zuschauer rät eine der drei Türen und der Showmaster Monty Hall wird daraufhin eine weitere Tür öffnen, hinter der sich aber kein Preis verbirgt. Der Zuschauer erhält jetzt die Möglichkeit, seine Wahl zu ändern. Sollte er dies tun? Wir betrachten die Ereignisse

- P_i, dass sich der Preis hinter Tür i befindet,
- Z_i, dass der Zuschauer zuerst Tür i wählt und
- M_i, dass Monty Hall Tür i nach der ersten Wahl des Zuschauers öffnet.

Wir nehmen o.B.d.A. an, dass der Zuschauer zuerst Tür 1 wählt und dass Monty Hall daraufhin Tür 2 öffnet; desweiteren nehmen wir an, dass der Zuschauer wie auch Monty Hall seine Wahl jeweils nach der Gleichverteilung trifft. Wir müssen die bedingten Wahrscheinlichkeiten prob[$P_1 \mid Z_1, M_2$] und prob[$P_3 \mid Z_1, M_2$] berechnen und beachten, dass prob[$P_1 \mid Z_1, M_2$] + prob[$P_3 \mid Z_1, M_2$] = 1 gilt, denn der Preis befindet sich nicht hinter der geöffneten Tür 2. Nach Definition der bedingten Wahrscheinlichkeiten ist

$$\begin{aligned} \text{prob}[\, P_1 \mid Z_1, M_2 \,] \cdot \text{prob}[\, Z_1, M_2] &= \text{prob}[\, Z_1, M_2 \mid P_1 \,] \cdot \text{prob}[\, P_1 \,] \text{ und} \\ \text{prob}[\, P_3 \mid Z_1, M_2 \,] \cdot \text{prob}[\, Z_1, M_2] &= \text{prob}[\, Z_1, M_2 \mid P_3 \,] \cdot \text{prob}[\, P_3 \,]. \end{aligned}$$

Wir bestimmen jeweils die rechten Seiten und erhalten

$$\text{prob}[\, P_1 \mid Z_1, M_2 \,] \cdot \text{prob}[\, Z_1, M_2] = (\frac{1}{3} \cdot \frac{1}{2}) \cdot \frac{1}{3},$$

denn Monty Hall kann die Türen 2 und 3 öffnen. Andererseits ist

$$\text{prob}[\ P_3 \mid Z_1, M_2\ ] \cdot \text{prob}[\ Z_1, M_2] = (\frac{1}{3} \cdot 1) \cdot \frac{1}{3},$$

denn Monty Hall kann nur Tür 2 öffnen. Also ist

$$\text{prob}[\ P_3 \mid Z_1, M_2\ ] \cdot \text{prob}[\ Z_1, M_2] = 2 \cdot \text{prob}[\ P_1 \mid Z_1, M_2\ ] \cdot \text{prob}[\ Z_1, M_2]$$

und wir erhalten prob[$P_3 \mid Z_1, M_2$] $= \frac{2}{3}$ und prob[$P_1 \mid Z_1, M_2$] $= \frac{1}{3}$: Der Zuschauer sollte seine Wahl stets ändern!

Eine kompaktere Argumentation ist wie folgt: Sei W die *Zufallsvariable*, die die erste Wahl des Zuschauers beschreibt und sei T die Zufallsvariable, die die richtige Tür beschreibt. Dann findet die Änderungsstrategie genau dann die richtige Tür, wenn W und T unterschiedliche Werte annehmen und dieses *Ereignis* hat die Wahrscheinlichkeit $\frac{2}{3}$. Fazit: Mit höherer Wahrscheinlichkeit war die erste Wahl schlecht und wird durch die veränderte Wahl richtig.

Definition 1.6 Zufallsvariablen $X_1, \ldots, X_n$ heißen unabhängig genau dann, wenn für alle $x_1, \ldots, x_n \in \mathbb{R}$

$$\text{prob}[\ X_1 = x_1 \wedge \cdots \wedge X_n = x_n\ ] = \Pi_{i=1}^{n} \text{prob}[\ X_i = x_i\ ].$$

Die Zufallsvariablen $X_1, \ldots, X_n$ heißen k-fach unabhängig, wenn $X_{i_1}, \ldots, X_{i_k}$, für jede Teilmenge $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ von k Elementen, unabhängig sind.

Beachte, dass wiederholte Experimente unabhängigen Zufallsvariablen entsprechen. Unabhängige Zufallsvariablen $X_1, \ldots, X_n$ sind k-fach unabhängig für jedes $k < n$. Die Umkehrung gilt im Allgemeinen nicht.

Aufgabe 3
Konstruiere Zufallsvariablen X_1, X_2, X_3, die 2-fach unabhängig, aber nicht unabhängig sind.

Aufgabe 4
Eine *stochastische Zufallsquelle* liefert Nullen und Einsen, wobei jedes Zeichen unabhängig von der Vorgeschichte ist und die Wahrscheinlichkeit für eine Eins konstant p ist. Wir haben Zugang zu einer solchen Zufallsquelle Q, kennen aber die Wahrscheinlichkeit p nicht. Unsere Aufgabe ist es aus dieser Zufallsquelle eine stochastische Zufallsquelle zu konstruieren, die eine 1 mit Wahrscheinlichkeit $\frac{1}{2}$ erzeugt. Wir erhalten also den Ausgabestrom von Q und müssen aus den Bits neue Zufallsbits generieren. Dabei soll die erwartete Zahl der zwischen zwei neu generierten Bits verarbeiteten Zeichen aus Q den Wert $\frac{1}{p \cdot (1-p)}$ nicht überschreiten.

Wie kann diese Konstruktion gelingen?

Hinweis: Betrachte zwei aufeinanderfolgende Bits von Q.

Beispiel 1.3 **Bestimmung des durchschnittlichen Gehalts ohne das eigene Gehalt preiszugeben.**
n Personen $1, \ldots, n$ wollen ihr durchschnittliche Gehalt berechnen, aber keiner möchte dabei sein eigenes Gehalt mitteilen. Wenn alle Personen ehrlich sind, dann ist das durchschnittliche Gehalt korrekt zu bestimmen; wenn hingegen irgendwelche k Personen unehrlich sind, dann sollten sie nicht mehr als die Summe der Gehälter der restlichen $n - k$ Personen in Erfahrung bringen können.

Algorithmus 1.7 **Ein Protokoll**

(1) M sei eine Zahl, von der bekannt ist, dass M größer als die Summe der Gehälter ist.
Jede Person i wählt zufällig Zahlen $X_{i,1}, \ldots, X_{i,i-1}, X_{i,i+1}, \ldots X_{i,n} \in \{0, \ldots, M-1\}$ und gibt $X_{i,j}$ an Person j weiter.

(2) Wenn G_i das Gehalt von Person i ist, dann berechnet i die Restklasse

$$S_i = G_i + \sum_{j,j\neq i}^{n} X_{j,i} - \sum_{j,j\neq i}^{n} X_{i,j} \bmod M.$$

Schließlich wird S_i bekannt gegeben.

(3) Jede Person i gibt S_i bekannt und berechnet dann $\sum_i S_i \bmod M$.

Kommentar: Wenn alle Personen ehrlich sind, dann ist

$$\sum_i S_i \bmod M \equiv \sum_i G_i + \sum_{i,j,j\neq i}^{n} X_{j,i} - \sum_{i,j,j\neq i}^{n} X_{i,j} \bmod M \equiv \sum_i G_i \bmod M = \sum_i G_i.$$

Also ist $\frac{1}{n} \cdot \sum_j S_j$ das gewünschte durchschnittliche Gehalt.

Warum ist dieses Protokoll sicher? Angenommen, die letzten k Personen sind Betrüger. Wir setzen $S_i^* = G_i + \sum_{j=1,j\neq i}^{n-k} X_{j,i} - \sum_{j=1,j\neq i}^{n-k} X_{i,j} \bmod M$. Eine ehrliche Person i veröffentlicht

$$S_i = S_i^* + \sum_{j=n-k+1}^{n} X_{j,i} - \sum_{j=n-k+1}^{n} X_{i,j} \bmod M.$$

Beachte, dass $\sum_{i=1}^{n-k} S_i^* = \sum_{i=1}^{n-k} G_i$.

Aufgabe 5
Zeige: Jede Wertekombination $(s_2^*, \ldots, s_{n-k}^*)$ der Zufallsvariablen $S_2^*, \ldots, S_{n-k}^*$ wird mit Wahrscheinlichkeit $M^{-(n-k-1)}$ ausgegeben. Also sind die Zufallsvariablen $S_2^*, \ldots, S_{n-k}^*$ unabhängig.

Als Konsequenz der Übungsaufgabe sind die Zufallsvariablen $S_2^*, \ldots, S_{n-k}^*$ unabhängig und gleichverteilt in $\{0, \ldots, M-1\}$. Man mache sich klar, dass diese Aussage auch für $S_2, \ldots, S_{n-k}$ gilt und die Betrüger lernen nichts, da jede Folge von $n-k-1$ Werten mit Wahrscheinlichkeit $\frac{1}{M^{n-k-1}}$ auftritt. Nur durch die Hinzunahme von S_1 können die Betrüger mit Hilfe der Beziehung $\sum_{i=1}^{n-k} S_i^* = \sum_{i=1}^{n-k} G_i$ das Gesamtgehalt der ehrlichen Personen bestimmen.

Lemma 1.8 *Seien X und Y Zufallsvariablen.*

(a) Es ist stets $E[\ X + Y\ ] = E[X] + E[Y]$. Wenn X und Y unabhängig sind, dann gilt $E[\ X \cdot Y\ ] = E[X] \cdot E[Y]$.

(b) Es ist stets $V[\ a \cdot X + b\ ] = a^2 \cdot V[\ X\ ]$.
Wenn X und Y unabhängig sind, dann ist $V[\ X + Y\ ] = V[\ X\ ] + V[\ Y\ ]$.

(c) Es seien V_1 und V_2 zwei Ereignisse. Dann ist stets

$$prob[\ V_1 \cup V_2\ ] \leq prob[V_1] + prob[V_2].$$

Wenn V_1 und V_2 unabhängige Ereignisse sind, dann ist

$$prob[\ V_1 \cap V_2\ ] = prob[V_1] \cdot prob[V_2].$$

Aufgabe 6
Zeige $E(X \cdot Y) = E(X) \cdot E(Y)$ für unabhängige diskrete Zufallsvariablen X und Y.

Aufgabe 7
Wir nehmen in einem Casino an einem Spiel mit Gewinnwahrscheinlichkeit $p = 1/2$ teil. Wir können einen beliebigen Betrag einsetzen. Geht das Spiel zu unseren Gunsten aus, erhalten wir den Einsatz zurück und zusätzlich denselben Betrag aus der Bank. Endet das Spiel ungünstig, verfällt unser Einsatz. Wir betrachten die folgende Strategie:

```
i:=0
REPEAT
    setze 2^i$
    i:=i+1
UNTIL(ich gewinne zum ersten mal)
```

Bestimme den erwarteten Gewinn dieser Strategie und die erwartete notwendige Liquidität (also den Geldbetrag, den man zur Verfügung haben muss, um diese Strategie ausführen zu können).

1.2.1 Abweichungen vom Erwartungswert

Die folgenden Abschätzungen sind besonders hilfreich, wenn Abweichungen vom Erwartungswert untersucht werden.

Lemma 1.9 *Sei X eine Zufallsvariable.*

(a) Die Markoff-Ungleichung für eine Zufallsvariable $X \geq 0$ and $a \in \mathbb{R}_{>0}$:

$$\text{prob}[\ X > a\ ] \leq \frac{E[X]}{a}.$$

(b) Die Tschebyscheff Ungleichung:

$$\text{prob}[\ |X - E[X]| > t\ ] \leq \frac{V[X]}{t^2}.$$

(c) Die Chernoff Ungleichungen: $X_1, \ldots, X_k$ seien unabhängige binäre Zufallsvariablen, wobei $p_i = prob[\ X_i = 1\ ]$ die Erfolgswahrscheinlichkeit von X_i ist. Dann ist $E = \sum_{i=1}^{k} p_i$ die erwartete Anzahl von Erfolgen und es gilt

$$\begin{aligned}
\text{prob}[\ \sum_{i=1}^{k} X_i > (1+\beta) \cdot E\ ] &\leq \left(\frac{e^{\beta}}{(1+\beta)^{1+\beta}}\right)^{E} \leq e^{-E \cdot \beta^2/3} \\
\text{prob}[\ \sum_{i=1}^{k} X_i < (1-\beta) \cdot E\ ] &\leq \left(\frac{e^{-\beta}}{(1-\beta)^{1-\beta}}\right)^{E} \leq e^{-E \cdot \beta^2/2}
\end{aligned}$$

für jedes $\beta > 0$ (bzw. $0 < \beta \leq 1$ im zweiten Fall).

Beweis (a): Nach Definition des Erwartungswerts ist

$$\begin{aligned}
E[X] &= \sum_{u \in U} \pi(u) \cdot X(u) \\
&\geq \sum_{u \in U, X(u) > a} \pi(u) \cdot X(u) \\
&\geq \text{prob}[\ X > a\ ] \cdot a.
\end{aligned}$$

(b) Wir wenden die Markoff-Ungleichung mit $a = t^2$ für die Zufallsvariable $(X - E[X])^2$ an und erhalten

$$\text{prob}[\ (X - E[X])^2 > t^2\ ] \leq \frac{E[(X - E[X])^2]}{t^2}.$$

Offensichtlich ist $|X - E[X]| > t \Leftrightarrow (X - E[X])^2 > t^2$ und die Behauptung folgt, wenn wir beachten, dass

$$\begin{aligned} E[\ (X - E[X])^2\ ] &= E[\ X^2 - 2X \cdot E[X] + E[X]^2\ ] = E[X^2] - 2E[X]^2 + E[X]^2 \\ &= E[X^2] - E[X]^2 = V[X]. \end{aligned}$$

(c) Wir zeigen nur, dass die erste Abschätzung gilt und stellen die zweite Abschätzung als Übungsaufgabe. Wir erhalten mit der Markoff-Ungleichung für beliebiges $\alpha > 0$

$$\begin{aligned} \text{prob}[\ \sum_{i=1}^{k} X_i > t\ ] &= \text{prob}[\ e^{\alpha \cdot \sum_{i=1}^{k} X_i} > e^{\alpha \cdot t}\ ] \\ &\leq e^{-\alpha \cdot t} \cdot E[e^{\alpha \cdot \sum_{i=1}^{k} X_i}] \\ &= e^{-\alpha \cdot t} \cdot \Pi_{i=1}^{k} E[e^{\alpha \cdot X_i}]. \end{aligned}$$

In der letzten Gleichung haben wir benutzt, dass $E[Y_1 \cdot Y_2] = E[Y_1] \cdot E[Y_2]$ gilt, wenn Y_1 und Y_2 unabhängige Zufallsvariablen sind. Wir ersetzen t durch $(1+\beta) \cdot E$, α durch $\ln(1+\beta)$ und erhalten

$$\begin{aligned} \text{prob}[\ \sum_{i=1}^{k} X_i > (1+\beta) \cdot E\ ] &\leq e^{-\alpha \cdot (1+\beta) \cdot E} \cdot \Pi_{i=1}^{k} E[e^{\alpha \cdot X_i}] \\ &= (1+\beta)^{-(1+\beta) \cdot E} \cdot \Pi_{i=1}^{k} E[(1+\beta)^{X_i}]. \end{aligned}$$

Die Behauptung folgt, da $E[(1+\beta)^{X_i}] = p_i(1+\beta) + (1-p_i) = 1 + \beta \cdot p_i \leq e^{\beta \cdot p_i}$. □

Aufgabe 8
Für $\beta \leq 1$ gilt stets $\frac{e^\beta}{(1+\beta)^{1+\beta}} = e^{\beta - (1+\beta)\ln(1+\beta)} \leq e^{-\beta^2/3}$.

Aufgabe 9
$X_1, \ldots, X_k$ seien unabhängige binäre Zufallsvariablen mit $p_i = \text{prob}[\ X_i = 1\ ]$. Es sei $E^* \geq \sum_{i=1}^{k} p_i$. Zeige, dass

$$\text{prob}[\ \sum_{i=1}^{k} X_i > (1+\beta) \cdot E^*\ ] \leq \left(\frac{e^\beta}{(1+\beta)^{1+\beta}}\right)^{E^*} \leq$$

für jedes $\beta > 0$ gilt.

Aufgabe 10
Zeige die zweite Chernoff Ungleichung.

Beispiel 1.4 Wir nehmen an, dass ein Zufallsexperiment X vorliegt, dessen Erwartungswert wir experimentell messen möchten. Das Experiment ist aber instabil, d.h. die Varianz von X ist groß.

Wir „boosten“, wiederholen also das Experiment k mal. Wenn X_i das Ergebnis des $i$ten Experiments ist, dann setzen wir $Y = \frac{1}{k} \cdot \sum_{i=1}^{k} X_i$ und beachten, dass die Zufallsvariablen $X_1, \ldots, X_k$ unabhängig sind: Es gilt also

$$V[\ Y\ ] = \frac{1}{k^2} \cdot V[\ \sum_{i=1}^{k} X_i\ ] = \frac{1}{k^2} \cdot \sum_{i=1}^{k} V[\ X_i\ ] = \frac{1}{k} \cdot V[\ X\ ].$$

Wir haben die Varianz um den Faktor k gesenkt, aber den Erwartungswert unverändert gelassen, denn $E[\ Y\ ] = E[\ \frac{1}{k} \cdot \sum_{i=1}^{k} X_i\ ] = \frac{1}{k} \cdot \sum_{i=1}^{k} E[\ X\ ] = E[\ X\ ]$. Die Tschebyscheff Ungleichung liefert jetzt das Ergebnis

$$\text{prob}[\ |Y - E[X]| > t\ ] = \text{prob}[\ |Y - E[Y]| > t\ ] \leq \frac{V[Y]}{t^2} = \frac{V[X]}{k \cdot t^2}$$

und große Abweichungen vom Erwartungswert sind unwahrscheinlicher geworden.

Angenommen, wir haben erreicht, dass Y mit Wahrscheinlichkeit mindestens $p = 1 - \varepsilon$ in ein „Toleranzintervall“ $T = [E[\ X\ ] - \delta, E[\ X\ ] + \delta\ ]$ fällt. Können wir p „schnell gegen 1 treiben“? Wir wiederholen diesmal das Experiment Y und zwar m mal und erhalten wiederum unabhängige Zufallsvariablen $Y_1, \ldots, Y_m$. Als Schätzung des Erwartungswerts geben wir jetzt den *Median* M von $Y_1, \ldots, Y_m$ aus. Wie groß ist die Wahrscheinlichkeit, dass M nicht im Toleranzintervall T liegt?

Y liegt mit Wahrscheinlichkeit mindestens p in T. Wenn also der Median außerhalb des Toleranzintervalls liegt, dann liegen mindestens $\frac{m}{2}$ Einzelschätzungen außerhalb, während nur $(1-p) \cdot m = \varepsilon \cdot m$ außerhalb liegende Einzelschätzungen zu erwarten sind. Wir wenden die Chernoff Ungleichung an und beachten, dass $(1 + \frac{1-2\cdot\varepsilon}{2\cdot\varepsilon}) \cdot \varepsilon \cdot m = \frac{m}{2}$. Also erhalten wir mit $\beta = \frac{1-2\cdot\varepsilon}{2\cdot\varepsilon}$

$$\text{prob}[\ M \notin T\ ] \leq e^{-\varepsilon \cdot m \cdot \beta^2/3} = e^{-(1-2\cdot\varepsilon)^2 \cdot m/(12\cdot\varepsilon)}$$

und die Fehlerwahrscheinlichkeit fällt negativ exponentiell, falls $\varepsilon < \frac{1}{2}$.

Aufgabe 11
Gegeben ist eine Menge S mit n paarweise verschiedenen Elementen. Der folgende randomisierte Algorithmus findet das k-kleinste Element.

Eingabe: k, S.

(1) Wähle zufällig ein Element y aus S gemäß der Gleichverteilung.

(2) Partitioniere $S \setminus \{y\}$ in zwei Mengen S_1 und S_2, so dass jedes Element in S_1 kleiner als y ist, und jedes Element in S_2 größer als y ist.

(3)
- Wenn $|S_1| = k - 1$, dann gib y aus.
- Wenn $|S_1| \geq k$, dann $S := S_1$ und gehe zu (1).
- Ansonsten setze $S := S_2$, $k := k - |S_1| - 1$ und gehe zu (1).

Zeige, dass die erwartete Laufzeit $O(n)$ ist.

Beachte, dass die Partitionierung in (2) $|S|$ Schritte benötigt. Wir nehmen an, dass für (1) ein geeigneter Zufallsgenerator vorhanden ist, der in konstanter Zeit ein Element auswählt.

1.2.2 Die Entropie

Wir betrachten die Entropie und die Kullback-Leibler Divergenz.

Definition 1.10 $q = (q_a \mid a \in \Sigma)$ und $r = (r_a \mid a \in \Sigma)$ seien zwei Verteilungen, d.h. es gilt $\sum_{a\in\Sigma} q_a = \sum_{a\in\Sigma} r_a = 1$ und $q_a, r_a \geq 0$ für jeden Buchstaben $a \in \Sigma$.

(a) $H(q) = -\sum_a q_a \log q_a$ ist die Entropie der Verteilung q.

(b) $D(q \mid\mid r) = \sum_a q_a \cdot \log \frac{q_a}{r_a}$ ist die Kullback-Leibler Divergenz (oder die relative Entropie) von q bezüglich r.

Das folgende Lemma zeigt, dass die Kullback-Leibler Divergenz als die Distanz zweier Verteilungen interpretiert werden kann.

Lemma 1.11 *Seien q und r zwei Verteilungen. Dann ist stets*

$$\sum_a q_a \cdot \log \frac{q_a}{r_a} \geq 0$$

und Gleichheit gilt genau dann, wenn $q = r$.

Beweis: Wir beachten zuerst, dass $\ln(x) \leq x - 1$ für alle x gilt und Gleichheit nur für $x = 1$ zutrifft. Warum? Die Tangente von $\ln(x)$ am Punkt $x = 1$ ist die Gerade $y = x - 1$ und diese Gerade liegt oberhalb der (konkaven) Funktion. Also folgt aus $\ln \frac{1}{x} \geq 1 - x$,

$$\begin{aligned}
\sum_a q_a \cdot \log \frac{q_a}{r_a} &= \frac{1}{\ln(2)} \cdot \sum_a q_a \cdot \ln \frac{q_a}{r_a} \\
&\geq \frac{1}{\ln(2)} \cdot \sum_a q_a \cdot (1 - \frac{r_a}{q_a}) = \frac{1}{\ln(2)} \cdot \sum_a (q_a - r_a) \\
&= \frac{1}{\ln(2)} \cdot (\sum_a q_a - \sum_a r_a) = 0.
\end{aligned}$$

Warum gilt Gleichheit genau dann, wenn $q_a = r_a$ für alle Buchstaben a? □

Teil I

Randomisierte Algorithmen

Kapitel 2

Entwurfsmethoden

Ein randomisierter Algorithmus A verfügt über alle Fähigkeiten eines konventionellen deterministischen Algorithmus, kann aber zusätzlich auch Zufallsbits anfordern: Der Algorithmus erhält nach einer Anforderung das Bit 0 (bzw. 1) mit Wahrscheinlichkeit $\frac{1}{2}$. Ein randomisierter Algorithmus wird also auf jeder Eingabe im Unterschied zu deterministischen Algorithmen verschiedenste Berechnungen in Abhängigkeit von den erhaltenen Zufallsbits ausführen. Wir führen die Akzeptanzwahrscheinlichkeit und die erwartete Laufzeit ein.

Definition 2.1 Sei A ein randomisierter Algorithmus und w eine Eingabe für P.

(a) Die Wahrscheinlichkeit einer Berechnung B von A ist $p_B = 2^{-k}$, falls die Berechnung B k Zufallsbits anfordert.

(b) A akzeptiert w mit Wahrscheinlichkeit p, falls

$$\sum_{B \text{ akzeptiert } w} p_B = p.$$

(c) Sei Zeit_B die Anzahl der Schritte einer Berechnung B. Die erwartete Laufzeit von A auf Eingabe w ist dann

$$\sum_{B \text{ ist eine Berechnung für Eingabe } w} p_B \cdot \text{Zeit}_B.$$

Wir unterscheiden die folgenden Typen randomisierter Algorithmen auf Grund ihres jeweiligen Akzeptanzverhaltens. Wir beschränken uns erst einmal auf Algorithmen für „Sprachen“, also auf Algorithmen, die Eingaben entweder akzeptieren oder verwerfen.

Definition 2.2 Sei L eine Sprache und A ein randomisierter Algorithmus.

- A ist ein **Las-Vegas-Algorithmus** für L, falls A nie irrt. Für alle Eingaben x muss also gelten

$$\begin{array}{lcll} x \in L & \Rightarrow & \text{prob}[\ A \text{ verwirft } x\ ] & = 0, \\ x \notin L & \Rightarrow & \text{prob}[\ A \text{ akzeptiert } x\ ] & = 0. \end{array}$$

- A ist ein Algorithmus mit **einseitig beschränktem Fehler** für L, falls für alle Eingaben x gilt

$$\begin{array}{lcll} x \in L & \Rightarrow & \text{prob}[\ A \text{ verwirft } x\ ] & < \frac{1}{2}, \\ x \notin L & \Rightarrow & \text{prob}[\ A \text{ akzeptiertt } x\ ] & = 0. \end{array}$$

- A ist ein Algorithmus mit **zweiseitig beschränktem Fehler** für L, falls für alle Eingaben x gilt

$$\begin{aligned} x \in L \quad &\Rightarrow \quad \text{prob}[\ A \text{ verwirft } x\ ] \quad < \tfrac{1}{3}, \\ x \notin L \quad &\Rightarrow \quad \text{prob}[\ A \text{ akzeptiert } x\ ] \quad < \tfrac{1}{3}. \end{aligned}$$

Die Fehlerschranke $\frac{1}{3}$ für Algorithmen mit zweiseitig beschränktem Fehler lässt sich durch mehrfache Wiederholung der Berechnung mit unabhängigen Münzwürfen und nachfolgender Mehrheitsentscheidung beliebig senken: Wenn wir einen Algorithmus (mit zweiseitigem Fehler höchstens $\frac{1}{3}$) k-mal wiederholen und die Ergebnissen $X_1, \ldots, X_k$ erhalten, dann sind die Zufallsvariablen X_i unabhängig. Bei korrektem Ergebnis 1 ist die Erfolgswahrscheinlichkeit einer Wiederholung mindestens $\frac{2}{3}$ und damit ist der Erwartungswert mindestens $\frac{2 \cdot k}{3}$. Also liefert die Chernoff-Schranke

$$\text{prob}[\ \sum_{i=1}^{k} X_i < \frac{k}{2}\ ] = \text{prob}[\ \sum_{i=1}^{k} X_i < (1 - \frac{1}{4}) \cdot \frac{2k}{3}\ ] \leq e^{-2 \cdot k/(2 \cdot 3 \cdot 16)} = e^{-k/48}$$

als Fehlerwahrscheinlichkeit. Durch relativ wenige Wiederholungen kann die inverse Fehlerwahrscheinlichkeit größer als die Anzahl der Atome im Universum gemacht werden: Ein „Sechser im Lotto“ ist wahrscheinlicher als ein jemals beobachteter Fehler.

Für Algorithmen mit einseitigem Fehler höchstens $\frac{1}{2}$ genügt eine einfachere Vorgehensweise: Für jede Eingabe x führt man k Berechnungen unabhängig voneinander aus und akzeptiert genau dann, wenn mindestens eine Berechnung akzeptiert. Offenbar gilt im Fall $x \in L$, dass wir mit Wahrscheinlichkeit $> 1 - 2^k$ akzeptieren, und für $x \notin L$ stets richtigerweise verwerfen. Mit der gleichen Vorgehensweise kann für Las-Vegas Algorithmen die Wahrscheinlichkeit der Ausgabe „?“ auf 2^{-k} nach k-maligem Wiederholen gesenkt werden.

Aufgabe 12

Gegeben sei ein randomisierter Algorithmus mit beidseitig beschränktem Fehler $\frac{1}{3}$ und erwarteter Laufzeit höchstens $T(n)$ für jede Eingabe der Länge n. Konstruiere einen randomisierten Algorithmus mit beidseitig beschränktem Fehler $\frac{1}{3}$, der für jede Eingabe der Länge n die (worst case) Laufzeit höchstens $c \cdot T(n)$ mit einer *möglichst kleinen* Konstante c besitzt.

Hinweis: Reduziere den Fehler zunächst durch Wiederholung und stelle dann sicher, dass die Laufzeit nicht größer als $c \cdot T(n)$ ist.

Wir betrachten randomisierte Algorithmen, weil sie in wichtigen Anwendungen schnellere Lösungen als deterministische Algorithmen liefern. Zudem ist ihre Kodierung oft sehr viel einfacher und kompakter (wie etwa für randomisierte Primzahltests). Während man vermutet, dass Randomisierung für sequentielle Algorithmen eine höchstens polynomielle Beschleunigung ergibt, sind randomisierte Algorithmen beweisbar besser als ihre deterministischen Kollegen, wenn die Algorithmen kein vollständiges Wissen der Eingabe besitzen (siehe den zweiten Teil des Skripts).

Die folgenden Methoden und Sichtweisen haben sich als wesentlich für den Entwurf randomisierter Algorithmen herausgestellt.

- Vermeidung von worst-case Eingaben.

  Ein randomisierter Algorithmus repräsentiert eine Klasse $\mathcal{A}$ deterministischer Algorithmen, da wir für jede Setzung der Zufallsbits einen deterministischen Algorithmus erhalten. Die umgekehrte Sichtweise erklärt ein wichtiges Entwurfsprinzip randomisierter Algorithmen: Angenommen, wir haben eine Klasse $\mathcal{A}$ deterministischer Algorithmen

konstruiert, so dass die meisten Algorithmen der Klasse für jede Eingabe „funktionieren“. Dann erhalten wir einen guten randomisierten Algorithmus durch die zufällige Wahl eines Algorithmus aus der Klasse $\mathcal{A}$; der gewählte Algorithmus wird hochwahrscheinlich die vorgegebene Eingabe nicht als worst-case Eingabe besitzen.

Ein Anwendungsbeispiel dieser Sichtweise ist der randomisierte Quicksort, denn die meisten Positionen liefern gute Pivots. Eine weitere Anwendung stellen randomisierte Primzahltests dar: Zusammengesetzte Zahlen besitzen viele *Zeugen*, und ein randomisierter Algorithmus versucht, einen dieser Zeugen zufällig zu erzeugen.

In einigen Fällen lässt das vorliegende Problem bestimmte Veränderungen der Eingabe zu. Wenn ein fixierter Algorithmus auf den meisten Eingaben „funktioniert“, dann erhalten wir einen randomisierten Algorithmus durch zufälliges Perturbieren der Eingabe. Als Beispiel betrachten wir wieder Quicksort und zwar die Quicksort-Variante, die stets den ersten Schlüssel als Pivot wählt. Das worst-case Verhalten dieser Variante ist zwar quadratisch, aber wir erhalten einen schnellen Algorithmus, wenn wir das Eingabe-Array zufällig permutieren. Der Routing Algorithmus in Beispiel 2.1.3 ist eine weitere Anwendung dieser Methode.

- Randomisierung in Konfliktsituationen.

  Ein randomisierter Algorithmus ist nicht ausrechenbar, solange ein Gegner keine Einsicht in die benutzten Zufallsbits hat. Wir erreichen durch Randomisierung somit „faire“, bzw. leichter gewinnbare Spiele. Wir haben eine erste Anwendung bereits mit Algorithmus 1.7 kennengelernt: Die fairen Personen haben sich dort durch Randomisierung gegen die Preisgabe ihres Gehalts gewehrt.

  On-line Algorithmen müssen ihre Entscheidungen nur aufgrund der bisher gesehenen Eingaben treffen, kennen aber zukünftige Eingaben nicht. Hier ist es ratsam, sich nicht auf eine Vorgehensweise festzulegen, sondern gegen den Gegner, nämlich gegen worst-case Eingaben in der Zukunft, zu randomisieren. Wir werden im zweiten Teil des Skripts eine Vielzahl von Anwendungen kennenlernen.

- Berechnung auf Stichproben.

  Statt ein Problem für eine große Datenmengen zu lösen, ist es manchmal ausreichend, eine genügend große Stichprobe zu ziehen, das Problem auf der sehr viel kleineren Stichprobe zu lösen und sodann zu extrapolieren.

  Random Walks liefern manchmal die Möglichkeit, charakteristische Stichproben zu bestimmen. Anwendungen werden wir in Simulated Annealing, der Volumenmessung konvexer Objekte und in der Lösung von 3-Sat Problemen kennenlernen.

- Fingerprinting.

  Angenommen, wir möchten feststellen, ob sich ein System in einem angestrebten Idealzustand befindet. Dazu nehmen wir einen „zufälligen Fingerabdruck“ des Systems und vergleichen ihn mit dem entsprechenden Fingerabdruck des Idealzustands. Es stellt sich heraus, dass die Berechnung sehr kurzer Fingerabdrücke ausreicht, um Gleichheit zu überprüfen: Randomisierte Gleichheitstests mit Fingerabdrücken sind wesentlich effizienter als konventionelle deterministische Tests.

Hashing ist eine weitere Anwendung von Fingerprinting, da wir den Hashwert eines Schlüssels als seinen Fingerabdruck auffassen können. Wir werden das universelle Hashing kennenlernen, das hochwahrscheinlich für **jede** Folge von Schlüsseln schnell ist. Hashing ist nicht nur eine erfolgreiche Methode in der Konstruktion guter Datenstrukturen, sondern hat auch eine Vielzahl algorithmischer Anwendungen wie etwa in der approximativen Feststellung der Anzahl verschiedener Schlüssel in Datenstromalgorithmen.

- Symmetry Breaking.

  In parallelen oder verteilten Algorithmen ist die Kommunikation zwischen Prozessoren sehr zeit-aufwändig. In einigen Fällen kann durch lokales zufälliges Rechnen ohne größere Kommunikation sichergestellt werden, dass die Prozessoren sich auf die Berechnung einer (von mehreren) Lösungen einigen: Die Prozessoren verhalten sich durch zufälliges Rechnen unterschiedlich und brechen ihre Symmetrie.

- Die probabilistische Methode.

  In einigen Fällen ist es sehr schwierig, Objekte mit „bemerkenswerten“ Eigenschaften nach deterministischen Vorschriften zu konstruieren. Wenn aber die Mehrheit der Objekte die Eigenschaft erfüllt, dann wird die *zufällige* Erzeugung eines Objekts die gewünschte Eigenschaft hochwahrscheinlich erfüllen. Hier tritt also das überraschende Phänomen auf, dass es schwierig sein kann, ein zu einer substantiellen Mehrheit gehörendes Objekt zu konstruieren: Zufällige Objekte sind schwer zu beschreiben.

  In der Kombinatorik findet man eine Vielzahl von Beispielen, wie etwa die Konstruktion dichter Graphen mit kleinen Cliquen oder die Konstruktion von Expander-Graphen. Expander-Graphen haben zahlreiche Anwendungen in fehler-toleranten Berechnungen oder in der Kodierung.

Aufgabe 13
Wie groß ist die erwartete Größe einer Clique in einem ungerichteten Graphen mit Kantenwahrscheinlichkeit $\frac{1}{2}$?

Aufgabe 14
Ein (α, N)-Expander ist ein bipartiter Graph $G = (U, V, E)$, so dass *jede* Teilmenge $W \subseteq U$ der Größe höchstens N mindestens $\alpha \cdot |W|$ Nachbarn besitzt.

Angenommen, ein zufälliger bipartiter Graph auf den Knotenmengen $U = \{1, \ldots, n\}$ und $V = \{n + 1, \ldots, 2n\}$ wird mit einer Kantenwahrscheinlichkeit von $\frac{d}{n}$ erzeugt. Welche Amplifizierung α wird für $N = \frac{n}{2}$ erreicht?

2.1 Vermeidung von Worst-Case Eingaben

Universelles Hashing und die randomisierte Pivotwahl in Quicksort sind zwei erste Beipiele für die Vermeidung von worst-case Eingaben. Hier beschreiben wir Anwendungen in der Auswertung von Spielbäumen, im Entwurf von Datenstrukturen für das Wörterbuchproblem, im Paket-Routing und im Testen von Primzahlen.

2.1.1 Die Auswertung von Spielbäumen

Wir betrachten ein Zwei-Personen Spiel mit den Spielern A und B. Spieler A beginnt und die Spieler ziehen alternierend bis eine Endsituation erreicht ist, die für einen der beiden Spieler gewinnend ist.

Jedes Zwei-Personen Spiel mit nur beschränkt langen Spielen besitzt einen Spielbaum T: Knoten mit geradem (bzw. ungeradem) Abstand von der Wurzel sind mit MAX (bzw. MIN) beschriftet. Die Blätter von T sind entweder mit Null oder Eins markiert, wobei eine Eins (bzw. Null) andeutet, dass Spieler A gewonnen (bzw. verloren) hat. Die Auswertung des Spielbaums wird rekursiv definiert:

- der Spielwert eines Blatts ist die Beschriftung des Blatts,
- der Spielwert eines MAX-Knotens v ist das Maximum der Spielwerte der Kinder von v und
- der Spielwert eines MIN-Knotens v ist das Minimum der Spielwerte der Kinder von v.

Offensichtlich besitzt der beginnende Spieler A genau dann einen gewinnenden Anfangszug, wenn die Wurzel den Spielwert Eins erhält.

Wir beschränken uns auf die speziellen Spielbäume T_k^t, wobei T_k^t ein vollständig k-ärer Baum der Tiefe $2t$ ist. Jeder innere Knoten von T_k^t besitzt also k Kinder und wir nehmen an, dass alle Spiele aus genau $2t$ Zügen bestehen. Jede der 2^n (mit $n := k^{2t}$) Wertezuweisungen kann als eine Wertezuweisung an die Blätter vorkommen.

Wir stellen uns zuerst die Frage, wieviele Blätter von einem deterministischen Algorithmus inspiziert werden müssen, um den Baum auswerten zu können. Dabei erlauben wir sogar unbeschränkte Ressourcen und zählen nur die Anzahl inspizierter Blätter. Trotzdem stellt sich heraus, dass jeder noch so mächtige deterministische Algorithmus im Worst-Case alle Blätter inspizieren muss.

Lemma 2.3 *Sei X ein deterministischer Algorithmus, der den Baum T_k^t durch Abfragen der Blattwerte bestimmt. Dann gibt es eine Wertezuweisung der Blätter, so dass X alle Blätter von T_k^t inspizieren muss.*

Beweis: Wir führen eine Induktion nach der Tiefe t durch. Für $t = 1$ besteht T_k^1 aus der MAX-Wurzel w, gefolgt von den k MIN-Kindern $w_1, \ldots, w_k$. Sei X ein deterministischer Algorithmus.

Wir betrachten die ersten, höchstens $n-1$ Nachfragen von X. Wir beantworten jede Nachfrage mit dem Spielwert Eins, solange nicht das letzte Kind eines MIN-Knotens w_i nachgefragt wird und halten damit das „Schicksal“ eines jeden MIN-Knotens solange wie möglich offen. Wird das letzte Kind von w_i nachgefragt, dann antworten wir mit Null und halten damit das Schicksal der MAX-Wurzel solange wie möglich offen. Offensichtlich muss X auch das letzte Blatt nachfragen und dessen Wert bestimmt den Spielwert der Wurzel.

Für den Induktionsschritt von t auf $t+1$ betrachten wir wieder die MAX-Wurzel w mit ihren MIN-Kindern $w_1, \ldots, w_k$ und den MAX-Enkeln $w_{i,1}, \ldots, w_{i,k}$. Beachte, dass jeder MAX-Enkel $w_{i,j}$ die Wurzel eines Baums T_k^t ist.

Auch diesmal betrachten wir nur die ersten, höchstens $n-1$ Nachfragen von X. Für Nachfragen nach Blättern im Baum von $w_{i,j}$ wenden wir das Antwortschema für T_k^t an und halten somit das Schicksal von $w_{i,j}$ solange wie möglich offen. Wird das letzte Blatt von $w_{i,j}$ nachgefragt, dann stellen wir sicher, dass $w_{i,j}$ den Spielwert Eins erreicht. Ist aber $w_{i,j}$ das letzte „offene“ Kind von w_i, dann erzwingen wir, dass $w_{i,j}$ und damit auch der MIN-Knoten w_i den Wert Null erhält.

Auch diesmal müssen alle Blätter nachgefragt werden und die letzte Nachfrage entschiedet den Spielwert der Wurzel. □

Wir stellen jetzt einen Las Vegas Algorithmus vor, der im Worst-Case wesentlich schneller als deterministische Algorithmen sein wird.

Algorithmus 2.4 **Eine Las Vegas Auswertung**

(1) Der Spielbaum wird rekursiv, beginnend mit der Wurzel $v = w$, ausgewertet:

(1a) Wenn v ein MAX-Knoten ist, dann bestimme ein Kind v^* von v zufällig und werte den Teilbaum mit Wurzel v^* aus.

Erhält man den Spielwert Eins, dann beende die Auswertung des MAX-Knotens v mit dem Ergebnis Eins. Erhält man den Spielwert Null, dann wiederhole Schritt (1a) für ein zufällig bestimmtes, noch nicht ausgewertetes Kind v^* solange, bis entweder der Spielwert Eins erreicht wird (und v den Spielwert Eins erhält) oder bis alle Kinder ausgewertet sind (und v den Spielwert Null erhält).

(1b) Wenn v ein MIN-Knoten ist, dann bestimme ein Kind v^* von v zufällig und werte den Teilbaum mit Wurzel v^* aus.

Erhält man den Spielwert Null, dann beende die Auswertung des MIN-Knotens v mit dem Ergebnis Null. Erhält man den Spielwert Eins, dann wiederhole Schritt (1b) für ein zufällig bestimmtes, noch nicht ausgewertetes Kind v^* solange, bis entweder der Spielwert Null erreicht wird (und v den Spielwert Null erhält) oder bis alle Kinder ausgewertet sind (und v den Spielwert Eins erhält).

(2) Der Spielwert der Wurzel wird ausgegeben.

Wie groß ist die erwartete Anzahl inspizierter Blätter von T_2^t? Es stellt sich heraus, dass wir die Anzahl abgefragter Blätter für *jede* Zuweisung von Spielwerten signifikant senken können.

Satz 2.5 *Jeder deterministische Algorithmus wird im Worst-Case alle $n = 2^{2 \cdot t} = 4^t$ Blätter von T_2^t inspizieren. Der Las Vegas Algorithmus 2.4 wird hingegen nur die erwartete Anzahl von höchstens $n^{\log_4 3} = 3^t$ Blättern inspizieren.*

Beweis: Wir argumentieren wieder induktiv und zeigen zuerst für T_2^1, dass im Erwartungsfall höchstens drei der vier Blätter nachzufragen sind.

Fall 1: Der Spielwert der MAX-Wurzel ist Null. Dies ist genau dann der Fall, wenn beide MIN-Kinder den Spielwert Null besitzen. Die erwartete Anzahl nachgefragter Blätter eines MIN-Kinds ist dann genau $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 = \frac{3}{2}$ und der Spielwert der Wurzel wird mit $\frac{3}{2} + \frac{3}{2} = 3$ erwarteten Fragen bestimmt.

Fall 2: Der Spielwert der MAX-Wurzel ist Eins. Dies ist genau dann der Fall, wenn mindestens ein MIN-Kind v den Spielwert Eins besitzt. Das Kind v wird mit Wahrscheinlichkeit $\frac{1}{2}$ als erstes gewählt und damit genügen in diesem Fall zwei Nachfragen. Also ist der Erwartungswert durch $\frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 4 = 3$ beschränkt. (Tatsächlich genügen $\frac{1}{2} \cdot 2 + \frac{1}{2} \cdot (2 + 3/2) = 11/4$ Anfragen, da nur dann beide Min-Kinder der Wurzel ausgewertet werden müssen, wenn der Geschwisterknoten v' von v den Spielwert Null besitzt. Dann gelingt aber eine Auswertung von v' mit der erwarteten Anzahl von 3/2 Anfragen.)

Der Induktionsschritt verläuft ähnlich: Wenn der Spielwert der MAX-Wurzel Null ist, dann besitzen beide Kinder den Spielwert Null. Die erwartete Anzahl nachgefragter Blätter eines MIN-Kinds ist dann höchstens $\frac{1}{2} \cdot 3^t + \frac{1}{2} \cdot (3^t + 3^t) = \frac{3}{2} \cdot 3^t$ und der Spielwert der Wurzel wird mit erwarteten 3^{t+1} Fragen bestimmt.

Wenn der Spielwert der MAX-Wurzel Eins ist, dann besitzt ein MIN-Kind v den Spielwert Eins. Mit Wahrscheinlichkeit $\frac{1}{2}$ wird v zuerst gewählt und dann genügt die erwartete Anzahl von höchstens $(3^t + 3^t) = 2 \cdot 3^t$ nachgefragten Blättern. Wird der Geschwisterknoten zuerst nachgefragt, dann ist die erwartete Anzahl offensichtlich durch $4 \cdot 3^{t+1}$ beschränkt. Insgesamt ist die erwartete Anzahl also auch diesmal durch $\frac{1}{2} \cdot 2 \cdot 3^t + \frac{1}{2} \cdot 4 \cdot 3^t = 3^{t+1}$ beschränkt. □

Aufgabe 15

Wir betrachten einen vollständigen ternären Baum. Jedes Blatt hat Tiefe h und jeder innere Knoten hat genau 3 Kinder. Der Baum hat $n = 3^h$ Blätter, die mit 0 oder 1 markiert sind. Ein innerer Knoten soll als Markierung die Markierung, die die Mehrheit seiner Kinder hat, erhalten. Gesucht ist die Markierung der Wurzel.

- Beschreibe einen randomisierten Algorithmus, der die Markierung der Wurzel ermittelt und in der Erwartung die Markierung von möglichst wenigen Blättern liest. Analysiere die erwartete Anzahl gelesener Blätter.
- Beweise, dass jeder nichtdeterministische Algorithmus zur Lösung des Problems mindestens $n \cdot \left(\frac{2}{3}\right)^h$ Blätter inspizieren muss.

2.1.2 Skip-Listen

Skip-Listen unterstützen die Wörterbuch-Operationen Insert, Delete und Lookup in erwartet logarithmischer Zeit. Skip-Listen werden als (nicht notwendigerweise binäre) Bäume implementiert, wobei man durch Einfügen in eine zufällig ausgewürfelte Schicht des Baums versucht, die Tiefe des Baumes nicht zu stark anwachsen zu lassen.

Angenommen, eine Skip-Liste S speichert die Schlüssel $x_1 < x_2 < \cdots < x_{n-1} < x_n$. Die Skip-Liste unterteilt die Schlüsselmenge in Schichten

$$L_0 = \{-\infty, \infty\} \subseteq L_1 \subseteq \cdots L_{t-1} \subseteq L_t = \{-\infty, x_1, \ldots, x_n, \infty\}.$$

Die Schicht $L_k = \{-\infty, x_{i_1}, \ldots, x_{i_s}, \infty\}$ wird in die Suchintervalle

$$[-\infty, x_{i_1}], [x_{i_1}, x_{i_2}], \ldots, [x_{i_{s-1}}, x_{i_s}], [x_{i_s}, \infty]$$

unterteilt. Der geordnete Baum T von S besitzt genau $s + 1$ Knoten der Tiefe k, die der Reihe nach mit den Suchintervallen von L_k markiert werden. Zusätzlich werden die Knoten gleicher Tiefe „gefädelt“, d.h. der Knoten mit Suchintervall $[x_{i_{m-1}}, x_{i_m}]$ zeigt auf den Knoten mit Suchintervall $[x_{i_m}, x_{i_{m+1}}]$.

Schließlich werden Baumkanten wie folgt eingesetzt: Wenn Knoten v mit dem Intervall I und Knoten w mit dem Intervall J markiert ist, dann wird die Kante (v, w) genau dann eingesetzt, wenn $J \subseteq I$ und Tiefe(w) =Tiefe(v)+1. Da ein Intervall der „Tiefe“ k möglicherweise in mehrere Intervalle der Tiefe $k + 1$ zerlegt wird, erhalten wir im Allgemeinen nicht-binäre gefädelte Bäume der Tiefe t.

Wir implementieren zuerst die Operation Lookup(x). Wir beginnen an der Wurzel und durchlaufen mit Hilfe der Fädelungszeiger die Suchintervalle der Kinder der Wurzel. Wenn x Randpunkt eines Intervalls ist, dann haben wir den Schlüssel gefunden; ansonsten setzten wir unsere Suche mit dem eindeutig bestimmten Intervall fort, das den Schlüssel x enthält.

Zur Ausführung der Operation Delete(x) führen wir zuerst die Operation Lookup(x) aus. Wenn x in Tiefe k gespeichert ist, dann werden wir zwei Intervalle $[y, x]$ und $[x, z]$ in Tiefe k finden. Wir verschmelzen die beiden Intervalle in das Intervall $[y, z]$ und müssen diesen Verschmelzungsprozess für die entsprechenden Knoten der Tiefen $k + 1, \ldots, t$ fortsetzen.

Schließlich müssen wir die die Operation Insert(x) beschreiben. Wir bestimmen zuerst die Schicht, in die x einzufügen ist, durch Zufallsentscheidung: Wir würfeln mit Erfolgswahrscheinlichkeit $\frac{1}{2}$ solange, bis wir den ersten Erfolg erhalten. Wir nehmen an, dass wir im $k$ten Versuch erfolgreich sind[1].

Fall 1: $k \leq t$. Wir fügen x in die Schicht L_{t-k+1} ein. Dazu führen wir die Operation Lookup(x) bis zur Tiefe $t-k+1$ aus. Wenn wir dort im Intervall $[y,z]$ „landen", dann zerlegen wir $[y,z]$ in die beiden Intervalle $[y,x]$ und $[x,z]$. Dieser Zerlegungsprozess ist jetzt für die entsprechenden Knoten der Tiefen $k+1,\ldots,t$ fortsetzen.

Fall 2: $k > t$. Wir erhöhen die Tiefe von t auf k und ersetzen deshalb die Wurzel durch eine Kette der Länge $k-t$. Der Schlüssel x ist sodann in das Kind der Wurzel über die beiden Intervalle $[-\infty,x]$ und $[x,\infty]$ einzufügen. Wie auch im Fall 1 ist dieser Zerlegungsprozess für alle nachfolgenden Schichten fortzusetzen.

Der Vorteil der Skip-Listen ist aus der Implementierung der drei Operationen ersichtlich: Keinerlei Wartungsarbeiten zur Beibehaltung einer Balancierung sind notwendig. Die Laufzeit der Operationen wird durch die Länge des Suchpfads dominiert, wobei allerdings in der Bestimmung der Länge auch die Fädelungszeiger mit in Betracht gezogen werden müssen, da einige Knoten des Baums einen hohen Grad besitzen können. Wir beginnen unsere Analyse aber mit der Bestimmung der Tiefe.

Lemma 2.6 *Wir fixieren eine beliebige Folge von n Lookup-, Insert- und Delete-Operationen. Dann ist*

$$\textit{prob}[\ \textit{Der Baum hat mehr als } \alpha \cdot \log_2 n \textit{ Schichten}\ ] \leq \frac{1}{n^{\alpha-1}}.$$

Beweis: Schichten werden für jede Insert-Operation, also bis zu n-mal zufällig ausgewürfelt. Sei X_i die Zufallsvariable, die die zufällig bestimmte Schicht bei der $i$ten Insert-Operation angibt. Wir erhalten

$$\text{prob}[\ X_i > T\ ] \leq 2^{-T}$$

und deshalb ist

$$\text{prob}[\ \max_i X_i > T\ ] \leq \frac{n}{2^T}.$$

Das Ergebnis folgt, wenn wir $T = \alpha \cdot \log_2 n$ einsetzen. □

Wir analysieren jetzt die erwartete Länge eines Suchpfades, wobei Fädelungszeiger mitzuzählen sind.

Lemma 2.7 *Nach einer beliebigen Folge von n Insert- und Delete-Operationen werde eine Lookup-Operation durchgeführt. Dann hat der Suchpfad, inklusive der Fädelungszeiger, eine höchstens logarithmische erwartete Länge.*

Beweis: Nach Durchführung der n Operationen möge die Skip-Liste die Schlüssel $x_1 < x_2 < \cdots < x_{n-1} < x_n$ speichern. Wir betrachten einen beliebigen Knoten v. Dann gehört v zu einer Schicht L_k und besitzt r rechte Geschwisterknoten mit den Intervallen $[y_1,y_2],\ldots,[y_r,y_{r+1}]$.

Wir schätzen zuerst die erwartete Zahl r der rechten Geschwisterknoten ab. Der linkeste Schlüssel y_1 wurde entweder in v (und $r > 0$) oder in einer höheren Schicht (und $r = 0$) eingefügt. Die Schlüssel $y_2,\ldots,y_r$ gehören alle zur Schicht L_k und der rechteste Schlüssel y_{r+1} gehört zur Schicht L_{k-1} oder einer höheren Schicht. Mit welcher Wahrscheinlichkeit

[1] Zur Erinnerung: L_t ist die letzte Schicht, also die Schicht der Blätter.

werden y_1 und die im Vergleich zu y_1 nächstgrößeren Schlüssel $y_2, \ldots, y_r$ alle in die Schicht L_k eingefügt?

Wir nehmen zuerst an, dass v ein Blatt ist, dass also $k = t$ gilt. Dann wird y_{r+1} mit Wahrscheinlichkeit $1/2$ in eine höhere Schicht, und die Schlüssel $y_1, y_2, \ldots, y_r$ werden mit Wahrscheinlichkeit 2^{-r} in Schicht L_t eingefügt. Also hat v mit Wahrscheinlichkeit $2^{-(r+1)}$ genau r rechte Geschwister und mit Wahrscheinlichkeit $1/2$ keine rechten Geschwister: Es ist

$$E[r] = \sum_{i=0}^{\infty} 2^{-(i+1)} \cdot i = \frac{1}{2} \cdot \sum_{i=0}^{\infty} 2^{-i} \cdot i = 1.$$

Offensichtlich nimmt die erwartete Anzahl rechter Geschwisterknoten ab, wenn v einer höheren Schicht angehört und wir erhalten $E[r] \leq 1$ für alle Knoten v. Also ist die erwartete Kinderzahl höchstens 2.

Wir wissen weiterhin, dass die Skip-Liste, mit einer Wahrscheinlichkeit von höchstens $\frac{1}{n^{\alpha-1}}$ mehr als $\alpha \cdot \log_2 n$ Schichten besitzt. Wenn die Skip-Liste eine Tiefe größer als $2 \log_2 n$ besitzt, dann geschieht dies mit einer Wahrscheinlichkeit von höchstens $1/n$; in diesem Fall ist die Länge des Suchpfads aber durch n beschränkt. Also gilt $E[$ Länge des Suchpfads $]$

$$\begin{aligned}
&\leq n \cdot \text{prob}[\text{ Die Skip-Liste hat mehr als } 2 \cdot \log_2 n \text{ Schichten }] + \\
&\quad E[\text{ Länge des Suchpfads} \mid \text{Die Skip-Liste hat höchstens } 2 \cdot \log_2 n \text{ Schichten }] \\
&\leq 1 + E[\text{ Länge des Suchpfads} \mid \text{Die Skip-Liste hat höchstens } 2 \cdot \log_2 n \text{ Schichten }] \\
&\leq 1 + 2 \cdot (2 \cdot \log_2 n) = 1 + 4 \cdot \log_2 n
\end{aligned}$$

und das war zu zeigen. □

Wir können jetzt unsere Analyse abschließen.

Satz 2.8 *n Insert-, Delete- oder Lookup-Operationen werden in erwarteter Zeit $O(n \cdot \log_2 n)$ durchgeführt.*

2.1.3 Message-Routing in Würfeln

Im Message-Routing Problem ist ein Prozessor-Netzwerk $G = (V, E)$ gegeben. Wir nehmen an, dass jeder Prozessor $v \in V$ ein Paket P_v zu einem Prozessor $\pi(v)$ schicken möchte und dass die Funktion $\pi : V \to V$ eine Permutation ist. Das Paket P_v ist über einen Weg von v nach $\pi(v)$ zu verschicken, wobei wir allerdings voraussetzen, dass eine Kante zu jedem Zeitpunkt nur ein Paket transportieren kann. Unser Ziel ist der Versand aller Pakete in möglichst kurzer Zeit.

Wir nennen einen Routing-Algorithmus *konservativ*, wenn der Weg eines jeden Pakets P_v nur vom Empfänger $\pi(v)$ abhängt. Diese Klasse von Algorithmen erscheint vernünftig, aber deterministische Algorithmen dieser Klasse werden scheitern.

Fakt 2.1 *[KKT] Sei G ein Netzwerk von n Prozessoren. Jeder Prozessor in G habe höchstens d Nachbarn. Sei weiterhin A ein konservativer, deterministischer Routing-Algorithmus. Dann gibt es eine Permutation π für die A mindestens $\Omega(\sqrt{\frac{n}{d}})$ Schritte benötigt.*

Beispiel 2.1 Wir betrachten den d-dimensionalen Würfel $W_d = (\{0,1\}^d, E_d)$[2]. Jeder Knoten $v \in \{0,1\}^d$ hat d Nachbarn, nämlich alle Knoten $w \in \{0,1\}^d$, die sich in genau einem Bit

[2]Eine Kante $\{u, v\}$ liegt genau dann in E_d, wenn u und v sich in genau einem Bit unterscheiden.

von v unterscheiden. Zu jedem konservativen, deterministischen Routing-Algorithmus gibt es somit eine Permutation π, für die mindestens $\Omega(\sqrt{\frac{2^d}{d}})$ Schritte benötigt werden. Aber je zwei Knoten können mit Wegen der Länge höchstens d verbunden werden, und deshalb sollte eine Lösung in Zeit $O(d)$ die Erwartungshaltung sein.

Deterministische konservative Algorithmen $\mathcal{A}$ können somit die hochgradige Vernetzung des Würfels nicht ausnutzen. Beispielshaft betrachten wir die Bit-Fixing Strategie $\mathcal{A}$, die ein Paket P_v vom Start v wie folgt zum Ziel w transportiert:

- P_v wird zuerst über die Kante bewegt, der das linkeste, in v und w unterschiedliche Bit entspricht.

- Dieses Verfahren wird solange wiederholt, bis der Empfänger w erreicht wird: Für $v = 1111$ und $w = 1000$ ist also $(1111, 1011, 1001, 1000)$ der Bit-Fixing Weg von v nach w.

Bit-Fixing versagt zum Beispiel für das Routing Problem $(x, y) \to (y, x)$, wobei x (bzw. y) der Block der ersten (bzw. zweiten) Hälfte der Bits des Senders ist. Was passiert? Die $\sqrt{2^d}$ Pakete der Sender $(x, 0)$ überschwemmen das Nadelöhr $(0, 0)$.

Randomisierte konservative Algorithmen haben hingegen für den Würfel keinerlei Probleme.

Algorithmus 2.9 **Randomisiertes Routing für den Würfel**

- Pakete seien auf dem d-dimensionalen Würfel gemäß der Permutation π zu verschicken.

- **Phase 1:** Versand an einen zufälligen Empfänger.

  Jedes Packet P_v wählt einen Empfänger $z_v \in \{0, 1\}^d$ zufällig. Das Paket P_v folgt sodann dem Bit-Fixing Weg von v nach z_v.

- **Phase 2:** Versand an den ursprünglichen Empfänger.

  Das Paket P_v folgt dem Bit-Fixing Weg von z_v nach $\pi(v)$.

Beachte, dass wir keinerlei Vorschriften für die Behandlung der den Kanten zugeordneten Warteschlangen zu folgen haben. Die einzige Bedingung ist, dass ein Paket über eine Kante zu verschicken ist, wenn mindestens ein Paket wartet.

Wir beginnen mit der Analyse der ersten Phase. Die folgenden Beobachtungen sind zentral.

- Wenn zwei Pakete P_v und P_w sich zu irgendeinem Zeitpunkt der ersten Phase trennen, dann werden sie sich in der ersten Phase nicht wieder treffen.

- Das Paket P_v laufe über den Weg $(e_1, \ldots, e_k)$. Sei $\mathcal{P}$ die Menge der von P_v verschiedenen Pakete, die irgendwann während der ersten Phase über eine Kante in $\{e_1, \ldots, e_k\}$ laufen. Die Wartezeit von P_v ist dann durch $|\mathcal{P}|$ beschränkt.

Die zweite Beobachtung ist nicht offensichtlich, da ein Paket P_w das Paket P_v wiederholt blockieren kann. Führe deshalb eine Induktion über die Größe von $\mathcal{P}$ aus. Der Basisfall $|\mathcal{P}| = 1$ ist trivial und wir nehmen $|\mathcal{P}| = k + 1$ an.

Aufgabe 16
Zeige, dass es ein Paket P_w gibt, das sich nur einmal in der selben Warteschlange mit P_v zusammen aufhält.

Dieses Paket P_w ist also nur für einen Wartezeittakt verantwortlich. Wir entfernen P_w aus $\mathcal{P}$ und erhalten die Behauptung aus der Induktionsvoraussetzung. Für die weitere Analyse betrachten wir die Zufallsvariablen

$$H_{v,w} = \begin{cases} 1 & P_v \text{ und } P_w \text{ laufen zumindest über eine gemeinsame Kante} \\ 0 & \text{sonst.} \end{cases}$$

Nach den obigen Beobachtungen ist die Gesamtwartezeit von P_v während der ersten Phase durch $\sum_w H_{v,w}$ beschränkt. Wir möchten die Chernoff-Schranke anwenden und bestimmen deshalb zuerst den Erwartungswert. Für die Kante e sei W_e die Anzahl der Bit-Fixing Wege, die über Kante e laufen. Wenn Paket P_v den Weg $(e_1, \ldots, e_k)$ einschlägt, dann ist

$$\begin{aligned} E[\sum_w H_{v,w}\,] &\leq E[\sum_{i=1}^{k} W_{e_i}\,] \\ &= k \cdot E[\,W_{e_1}\,], \end{aligned}$$

denn jede Kante hat dieselbe Belastung zu erwarten. Wir können die erwartete Belastung $E[\,W_e\,]$ der Kante e wie folgt bestimmen: Die erwartete Weglänge eines jeden Pakets ist $\frac{d}{2}$, denn Start und Ziel werden sich im Erwartungsfall in genau der Hälfte aller Bitpositionen unterscheiden. Die $d \cdot 2^{d-1}$ Kanten des Würfels tragen alle dieselbe Belastung und

$$E[\,W_e\,] = \frac{\frac{d}{2} \cdot 2^d}{d \cdot 2^{d-1}} = 1$$

folgt. $\frac{d}{2}$ ist die erwartete Weglänge von P_v und deshalb ist

$$E[\sum_w H_{v,w}\,] \leq \frac{d}{2}.$$

Wir wenden die Chernoff-Schranke an und erhalten

$$\text{prob}[\sum_w H_{v,w} \geq (1+\beta) \cdot \frac{d}{2}\,] \leq e^{-\beta^2 \cdot \frac{d}{2}/3}.$$

Für $\beta = 3$ ist also die Wahrscheinlichkeit einer Gesamtwartezeit für P_v von mindestens $2d$ durch $e^{-3 \cdot d/2}$ beschränkt. Da wir insgesamt 2^d Pakete besitzen, ist damit auch die Wahrscheinlichkeit, dass *irgendein* Paket länger als $2d$ Schritte wartet, durch $2^d \cdot e^{-3 \cdot d/2} \leq e^{-d/2}$ beschränkt. Die Analyse der zweiten Phase ist völlig analog und wir können zusammenfassen.

Satz 2.10 *Die folgenden Aussagen gelten mit Wahrscheinlichkeit mindestens* $1 - 2 \cdot e^{-d/2}$*:*

(a) Jedes Paket wird in jeder der beiden Phasen nach höchstens $2d$ *Schritten sein Ziel erreichen.*

(b) Nach höchstens $4d$ *Schritten ist jedes Paket am Ziel.*

2.1.4 Primzahltests

Das RSA-Kryptosystem verlangt zwei „geheime“, große Primzahlen p und q, die nur der Empfängerin Alice bekannt sind. Alice gibt dann das Produkt $N = p \cdot q$ sowie einen Kodierungsexponenten e bekannt; daraufhin kann Bob seine Nachricht x gemäß $y = x^e \bmod N$ verschlüsseln.

Sei $\phi(N) = |\{1 \leq x \leq N-1 \| \ (x, N) = 1\}|$ die Anzahl der primen Restklassen modulo N. Die Berechnung von $\phi(N)$ ist für Außenstehende sehr schwer, während Alice natürlich weiß, dass $\phi(N) = (p-1) \cdot (q-1)$ ist. Sie berechnet das multiplikative Inverse d von e modulo $\phi(N)$ und dekodiert durch

$$y^d \equiv (x^e)^d \equiv x^{e \cdot d \bmod \phi(N)} \equiv x \mod N.$$

Kritisch für das RSA-Kryptosystem ist die Beschaffung großer, geheimer Primzahlen. Sei $\pi(x) = |\{p \leq x \mid p \text{ ist eine Primzahl } \}$. Nach dem Primzahlsatz ist

$$\frac{x}{\pi(x)} = \ln(x) + o(\ln(x))$$

und Primzahlen haben eine logarithmische Dichte: Um eine Primzahl mit B Bits zufällig auszuwürfeln, müssen im Erwartungsfall $O(B)$ Zufallszahlen mit B Bits auf ihre Primzahleigenschaft untersucht werden.

Damit ist die Beschaffung von zufälligen Primzahlen mit Zehntausenden von Bits eine Standardaufgabe, solange schnelle Primzahltests zur Verfügung stehen. Wir besprechen den randomisierten Primzahltests von Miller und Rabin. (Seit 2002 gibt es auch einen effizienten deterministischen Primzahltest [AKS], allerdings ist das neue Verfahren wesentlich langsamer als die unten beschriebenen randomisierten Verfahren.) Das Miller-Rabin Verfahren setzt den Fermat-Test ein, der auf der folgenden gruppentheoretischen Eigenschaft basiert.

Fakt 2.2 *Sei G eine endliche Gruppe.*

(a) Ist H eine Untergruppe von G, dann ist $|H|$ ein Teiler von $|G|$.

(b) Der kleine Satz von Fermat: Für jedes Element $g \in G$ ist

$$g^{|G|} = 1.$$

Aufgabe 17
Beweise den kleinen Satz von Fermat mit Hilfe von Teil (a).

Angenommen N ist eine Primzahl. Dann ist $\mathbb{Z}_N^*$, die Menge der primen Restklassen modulo N, eine Gruppe mit $N-1$ Elementen. Also ist der „Fermat-Test“

$$a^{N-1} \equiv 1 \bmod N$$

für jede Restklasse a erfüllt.

Beispiel 2.2 Die Zahl N heißt Carmichael-Zahl, wenn

$$a^{N-1} \equiv 1 \mod N$$

für jede mit N teilerfremde Zahl a gilt. Leider gibt es unendlich viele Carmichael-Zahlen, die keine Primzahlen sind. Zum Beispiel sind 561, 1105, 1729, 2465, 2821 solche „Möchtegern-Primzahlen“. Der einfache Fermat-Test genügt also für eine fehlerfreie Primzahl-Erkennung nicht.

Wenn N eine Primzahl ist, dann ist insbesondere $N-1$ durch 2 teilbar und $\mathbb{Z}_N$ ist ein Körper. Aber die quadratische Gleichung $x^2 = 1$ besitzt über jedem Körper nur die Lösungen $x \in \{-1, 1\}$, und es muss

$$a^{(N-1)/2} \equiv -1 \bmod N \quad \text{oder} \quad a^{(N-1)/2} \equiv 1 \bmod N$$

für jede Restklasse a gelten, *solange* N eine Primzahl ist. Wir formulieren jetzt den erweiterten Fermat Test für eine Restklasse a. Es gelte $N - 1 = 2^k \cdot m$ für eine ungerade Zahl m. Wenn $r < k$, dann muss aus den gleichen Gründen

$$a^{(N-1)/2^r} \equiv 1 \bmod N \quad \Rightarrow \quad a^{(N-1)/2^{r+1}} \equiv -1, 1 \bmod N \tag{2.1}$$

gelten, solange N eine Primzahl ist. Im **erweiterten Fermat Test** für eine Restklasse a überprüfen wir, ob $a^{N-1} \equiv 1 \mod N$ gilt und ob Eigenschaft (2.1) für jedes r mit $0 \leq r < k$ gilt. Der Miller-Rabin Primzahltest setzt darauf, dass eine zusammengesetzte Zahl N den erweiterten Fermat Test für die *meisten* Restklassen a mod N nicht besteht.

Algorithmus 2.11 **Der Miller-Rabin Primzahltest.**

(1) N sei die Eingabezahl und k sei maximal mit $N - 1 = 2^k \cdot m$.

(2) Wähle eine Restklasse $a \in \mathbb{Z}_N$ mit $a \not\equiv 0 \bmod N$ zufällig. Wenn $\mathrm{ggT}(a, N) \neq 1$, dann ist N zusammengesetzt.

(3) Klassifiziere N als zusammengesetzt, wenn N gerade ist oder

- $N = a^b$ für natürliche Zahlen $a, b \geq 2$ oder
- N den erweiterten Fermat Test für Restklasse a nicht besteht.

(4) Wenn N nicht als zusammengesetzt klassifiziert wurde, dann klassifiziere N als Primzahl.

Satz 2.12 *Der Miller-Rabin Primzahltest ist fehlerfrei, wenn eine Zahl N als zusammengesetzt klassifiziert wird. Ist N zusammengesetzt, dann wird N mit Wahrscheinlichkeit höchstens $\frac{1}{2}$ als Primzahl klassifiziert. Höchstens $O(\log_2 N)$ arithmetische Operationen werden durchgeführt.*

Beweis: Der Miller-Rabin Test ist offensichtlich fehlerfrei, wenn N als zusammengesetzt klassifiziert wird. Wir brauchen also nur den Fall der Klassifikation einer zusammengesetzten Zahl N als Primzahl betrachten.

Für jede Restklasse b modulo N sei $\mathrm{ord}_N(b)$ der kleinste Exponent r, so dass $b^r \equiv 1 \bmod N$. Die Menge $\{b, b^2, \ldots, b^{r-1}, b^r = 1\}$ definiert eine Untergruppe von $\mathbb{Z}_N^*$ und deshalb ist $r = \mathrm{ord}_N(b)$ ein Teiler der Gruppenordnung $|\mathbb{Z}_N^*|$ (siehe Fakt 2.2 (a)).

> Sei h die größte Zahl, so dass 2^{h+1} ein Teiler von $\mathrm{ord}_N(b)$ für mindestens eine Restklasse b ist.

Offensichtlich ist $(N-1)^2 \equiv 1 \bmod N$ und deshalb ist $\mathrm{ord}_N(N-1) = 2$. Wir können also $h \geq 0$ annehmen, da mindestens eine Restklasse die Ordnung zwei hat.

Wir erinnern daran, dass $N - 1 = 2^k \cdot m$ mit der ungeraden Zahl m gilt. Die folgende Beobachtung ist wichtig.

Lemma 2.13 *Wenn $a^{N-1} \equiv 1 \mod N$, dann ist $a^{2^{h+1} \cdot m} \equiv 1 \mod N$.*

Beweis: Sei $\mathrm{ord}_N(a) = 2^j \cdot m'$ für eine ungerade Zahl m'. Da wir $a^{N-1} \equiv 1 \mod N$ annehmen, ist also $2^j \cdot m'$ ein Teiler von $2^k \cdot m = N - 1$ und insbesondere muss m' ein Teiler von m sein. Desweiteren, nach Definition von h ist $j \leq h + 1$: Insgesamt ist also $\mathrm{ord}_N(a) = 2^j \cdot m'$ ein Teiler von $2^{h+1} \cdot m$. Dann muss aber $a^{2^{h+1} \cdot m} \equiv 1 \mod N$ gelten. □

Um den Fehler des Miller-Rabin Algorithmus abzuschätzen, betrachten wir

$$G = \{a \in \mathbb{Z}_N^* \mid a^{2^h \cdot m} \equiv -1, 1 \bmod N\}.$$

G ist eine Untergruppe von $\mathbb{Z}_N^*$, denn $(a \cdot b)^{2^h \cdot m} = a^{2^h \cdot m} \cdot b^{2^h \cdot m} = \pm 1 \cdot \pm 1 = \pm 1$ und G ist unter Multiplikation abgeschlossen. Desweiteren, wenn b das multiplikative Inverse von $a \in G$ ist, dann ist $1 \equiv (a \cdot b)^{2^h \cdot m} \equiv a^{2^h \cdot m} \cdot b^{2^h \cdot m} \equiv \pm 1 \cdot b^{2^h \cdot m}$ und b gehört zu G. Warum ist G von Interesse?

Lemma 2.14 *Wenn im Schritt (1) eine Restklasse $a \notin G$ ausgewürfelt wird, dann wird N als zusammengesetzt klassifiziert.*

Beweis: Wenn die zufällig ausgewürfelte Restklasse a nicht in G liegt, dann erhalten wir insbesondere $a^{2^h \cdot m} \not\equiv -1, 1 \mod N$. Wenn $a^{2^k \cdot m} \not\equiv 1 \mod N$, dann wird N als zusammengesetzt klassifiziert, und wir können deshalb $a^{2^k \cdot m} \equiv 1 \mod N$ annehmen. Mit Lemma 2.13 ist aber $a^{2^{h+1} \cdot m} \equiv 1 \mod N$: Da $a^{2^h \cdot m}$ die Wurzel von $a^{2^{h+1} \cdot m}$ ist und wir wissen, dass $a^{2^h \cdot m} \not\equiv -1, 1 \mod N$ gilt, haben wir N als zusammengesetzt enttarnt! □

Jetzt bleibt die zentrale Frage nach der Größe von G. Da G eine Untergruppe von $\mathbb{Z}_N^*$ ist, ist die Größe von G ein Teiler der Gruppenordnung von $\mathbb{Z}_N^*$. Wenn G von $\mathbb{Z}_N^*$ verschieden ist, dann wird somit eine zusammengesetzte Zahl mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ entlarvt. (Da $N \neq a^b$ nach Schritt (2) für jedes a und b gilt, ist N insbesondere keine Primzahlpotenz, und wir können annehmen, dass $N = x \cdot y$ für teilerfremde Zahlen x und y gilt.)

Lemma 2.15 *Es sei $N = x \cdot y$ mit teilerfremden Zahlen x und y. Dann ist $G \neq \mathbb{Z}_N^*$.*

Beweis: Wir wählen die Restklasse b, so dass 2^{h+1} ein Teiler von $\mathrm{ord}_N(b)$ ist. Wir haben $N = x \cdot y$ mit $\mathrm{ggT}(x, y) = 1$ angenommen.

Aufgabe 18
Sei $N = x \cdot y$ mit teilerfremden Zahlen x und y. Dann ist $\mathrm{ord}_N(b)$ das kleinste gemeinsame Vielfache von $\mathrm{ord}_x(b)$ und $\mathrm{ord}_y(b)$ für jede Restklasse b modulo N.

Wir können also o.B.d.A annehmen, dass 2^{h+1} ein Teiler von $\mathrm{ord}_x(b)$ ist. Wir bestimmen die Restklasse a modulo N mit Hilfe des Chinesischen Restsatzes so, dass

$$a \equiv b \bmod x \quad \text{und} \quad a \equiv 1 \bmod y$$

und behaupten, dass $a \notin G$. Wir beachten zuerst, dass

$$a^{2^h \cdot m} \equiv b^{2^h \cdot m} \not\equiv 1 \bmod x$$

gilt. Denn, wenn $b^{2^h \cdot m} \equiv 1 \bmod x$, dann ist $\mathrm{ord}_x(b)$ ein Teiler von $2^h \cdot m$. Aber wir wissen bereits, dass 2^{h+1} ein Teiler von $\mathrm{ord}_x(b)$ ist, und deshalb wäre 2 ein Teiler von m: Ein Widerspruch zur Konstruktion von m. Schließlich ist

$$a^{2^h \cdot m} \equiv 1 \bmod y,$$

denn $a \equiv 1 \bmod y$. Wenn $a^{2^h \cdot m} \not\equiv 1 \bmod x$ und $a^{2^h \cdot m} \equiv 1 \bmod y$, dann kann aber nicht $a^{2^h \cdot m} \equiv -1, 1 \bmod N$ gelten. □

Damit ist die Analyse des Miller-Rabin Tests abgeschlossen. □

2.2 Fingerprinting

Was ist ein Fingerabdruck? Angenommen, wir haben ein Universum U von Schlüsseln. Dann definiert jede Funktion

$$h : U \to \{0, \ldots, N\}$$

den Wert $h(x)$ als Fingerabdruck von x. Natürlich gilt $h(x) = h(y)$ wann immer x und y identisch sind. Wenn N sehr viel kleiner als die Mächtigkeit von U ist, dann muss aber auch zwangsläufig $h(x) = h(y)$ für verschiedene Schlüssel x und y gelten. Was aber passiert, wenn wir nicht mit einer Funktion h, sondern mit einer Klasse H von Hashfunktionen arbeiten und einen zufälligen Fingerabdruck durch die zufällige Wahl einer Funktion $h \in H$ nehmen?

2.2.1 Ein randomisierter Gleichheitstest

Gegeben sind zwei Polynome p und q in n Veränderlichen, wobei die Polynome als „Black-Box“ vorliegen: Die Black-Box für p (bzw. q) wird für Eingabe x die Ausgabe $p(x)$ (bzw. $q(x)$) bestimmen. Wir sollen entscheiden, ob $p \neq q$ ist. Dabei nehmen wir an, dass wir den maximalen Grad d von p und q kennen, wobei der Grad eines Polynoms

$$p(x_1, \ldots, x_n) = \sum_{(i_1,\ldots,i_n) \in \mathbb{N}_0^n} c_{(i_1,\ldots,i_n)} \cdot x_1^{i_1} \cdots x_n^{i_n}$$

in n Variablen $x_1, \ldots, x_n$ durch

$$\operatorname{grad}(p) := \max\{i_1 + \cdots + i_n \mid c_{(i_1,\ldots,i_n)} \neq 0\}$$

definiert ist. Beispielsweise gilt

$$\operatorname{Grad}(3x_1^3x_2 - x_3 + x_1x_2x_3 + 7x_2^3x_3^3) = \max\{4, 1, 3, 6\} = 6.$$

Der folgende Algorithmus löst das Gleichheitsproblem mit einseitigem Fehler für einen beliebigen Körper K.

Algorithmus 2.16

(0) Die Polynome p und q in n Veränderlichen sind durch jeweils eine Black-Box gegeben. Der Grad beider Polynome sei durch d nach oben beschränkt.

Kommentar: Der Algorithmus soll genau dann akzeptieren, wenn $p \not\equiv q$.

(1) Sei S eine beliebige Menge von $2d$ Körperelementen.

(2) Wähle k Vektoren $x^{(1)}, \ldots, x^{(k)} \in S^n$ zufällig gemäß der Gleichverteilung.

(3) Akzeptiere, wenn $p\big(x^{(i)}\big) \neq q\big(x^{(i)}\big)$ für mindestens ein i und verwirf sonst.

Offensichtlich macht Algorithmus 2.16 keinen Fehler, wenn akzeptiert wird. Wie groß ist der Fehler beim Verwerfen?

Satz 2.17 *Sei K ein Körper und sei $S \subseteq K$ eine endliche Menge. Wenn $x \in S^n$ gemäß der Gleichverteilung gewählt wird, dann gilt*

$$\text{prob}[\, r(x) = 0 \,] \leq \frac{d}{|S|}$$

für jedes vom Nullpolynom verschiedene Polynom r vom Grad d.

Beweis: Wir führen einen induktiven Beweis über die Dimension n. Wenn $n = 1$, dann ist r ein einstelliges Polynom. Das Polynom r ist vom Grad d und verschwindet deshalb auf höchstens d der $|S|$ Elemente von S.

Für den Induktionsschritt ziehen wir die Variable x_1 heraus und erhalten

$$r(x) = \sum_{i=0}^{d^*} x_1^i \cdot r_i(x_2, \ldots, x_{n+1}),$$

wobei $d^* \leq d$ maximal mit der Eigenschaft $r_{d^*} \not\equiv 0$ sei.

Nach Induktionsannahme wissen wir, dass $r_{d^*}(x_2, \ldots, x_{n+1}) = 0$ mit Wahrscheinlichkeit höchstens $(d - d^*)/|S|$ gilt. Wenn andererseits $r_{d^*}(x_2, \ldots, x_{n+1}) \neq 0$ ist, dann betrachten wir das einstellige Polynom

$$R(x_1) = \sum_{i=0}^{d^*} x_1^i \cdot r_i(x_2, \ldots, x_{n+1})$$

vom Grad d^*. Da $r_{d^*}(x_2, \ldots, x_{n+1}) \neq 0$ nach Annahme, stimmt R nicht mit dem Nullpolynom überein, und deshalb ist $R(x_1) = 0$ mit Wahrscheinlichkeit höchstens $d^*/|S|$. Insgesamt erhalten wir also

$$\begin{aligned} \text{prob}[\, r(x) = 0 \,] &\leq \text{prob}[\, r(x) = 0 \mid r_{d^*}(x_2, \ldots, x_{n+1}) \neq 0 \,] + \text{prob}[\, r_{d^*}(x_2, \ldots, x_{n+1}) = 0 \,] \\ &\leq \frac{d^*}{|S|} + \frac{d - d^*}{|S|} = \frac{d}{|S|} \end{aligned}$$

und das war zu zeigen. □

In Anwendungen wählt man zum Beispiel eine beliebige Teilmenge S von $2 \cdot d$ Körperelementen. Wenn fälscherlicherweise verworfen wird, dann ist $p - q$ vom Nullpolynom verschieden, aber k-maliges zufälliges unabhängiges Ziehen von Vektoren führt stets auf Nullstellen. Wenn wir $r := p - q$ setzen, erhalten wir dafür höchstens eine Wahrscheinlichkeit von 2^{-k}. Folglich besitzt unser Algorithmus einen einseitigen Fehler von höchstens 2^{-k}.

Aufgabe 19

Gegeben sind drei $n \times n$ Matrizen A, B, C mit Einträgen aus $\mathbb{Z}$. Es soll entschieden werden, ob $AB = C$. Gib einen randomisierten Algorithmus an, der das Problem in Zeit $O(n^2)$ mit beschränktem Fehler löst. Dabei sind Additionen und Multiplikationen in Zeit $O(1)$ erlaubt.

Aufgabe 20

Beim Pattern Matching Problem sind ein binärer Text der Länge n sowie ein binäres Muster der Länge $m < n$ gegeben. Es soll eine Stelle im Text gefunden werden, an der das Muster im Text auftritt, falls eine solche existiert.

Gib einen Algorithmus an, der das Problem in Zeit $O(n)$ auf einer probabilistischen Registermaschine löst. Wir nehmen dabei an, dass arithmetische Operationen für Operanden mit $O(\log_2 n)$ Bits in einem Schritt ausgeführt werden.

HINWEIS: Sei p eine zufällige Primzahl aus $\{1, \ldots, n^4\}$. Für feste, aber beliebige $x \neq y \in \{0,1\}^m$ ist die Wahrscheinlichkeit, dass Zahl$(x) \equiv$ Zahl(y) mod p, kleiner als $1/n$.

Aufgabe 21

$G = (V \cup W, E)$ mit $n := \mid V \mid = \mid W \mid$ sei ein bipartiter, ungerichteter Graph. $A = [a_{vw}]_{1 \leq v,w \leq n}$ sei eine Matrix, die Unbestimmte $x_{v,w}$ enthält, wobei

$$a_{vw} = \begin{cases} x_{vw} & \{v,w\} \in E \\ 0 & \text{sonst.} \end{cases}$$

1. Zeige: G besitzt ein perfektes Matching genau dann, wenn die Determinante von A nicht das Nullpolynom ist. Hinweis: Wenn S_n die Gruppe aller Permutationen bezeichnet, ist die Determinante durch

   $$\det(A) = \sum_{\sigma \in S_n} \text{signum}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

   gegeben.

2. Entwirf einen polynomiellen Algorithmus, der auf Determinatenberechnung basiert, und entscheidet, ob G ein perfektes Matching besitzt. Falls der Algorithmus ein perfektes Matching behauptet, muss die Antwort stimmen. Entscheidet der Algorithmus, dass ein perfektes Matching nicht existiert, so muss die Antwort mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ stimmen.

   Hinweis: Die Berechnung des Wertes einer Determinaten einer Matrix mit reellen Einträgen gelingt in polynomieller Zeit. Daher ist dieser Ansatz Grundlage schneller paralleler Algorithmen für Matching- und Flussprobleme.

3. Zeige, wie man durch einen effizienten randomisierten Algorithmus ein perfektes Matching bestimmt, wenn ein perfektes Matching existiert.

Aufgabe 22

Zwei Prozessoren A und B sind gegeben. A besitzt eine Eingabe $x \in \{0,1\}^n$, B eine Eingabe $y \in \{0,1\}^n$. Es ist zu entscheiden, ob $x = y$. Die Prozessoren haben *unbeschränkte Rechenkraft*, kennen aber die Eingabe des jeweils anderen Prozessors nicht. A kann jedoch (deterministisch oder randomisiert) eine von x abhängende Nachricht

$$nachricht(x) \in \{0,1\}^*$$

an B senden. B muss dann anhand dieser Nachricht und der eigenen Eingabe entscheiden, ob $x = y$ gilt. Die Vereinbarung, wie A zur zu sendenden Nachricht kommt und wie B aus Nachricht und seiner Eingabe die Antwort ermittelt, nennt man *Protokoll*.

1. Zeige, dass jedes deterministische Protokoll im worst case $\geq n$ Bits versenden muss.

2. Entwirf ein randomisiertes Protokoll, das $O(\log_2(n))$ Bits sendet. Eingaben mit $x = y$ müssen in jedem Fall akzeptiert werden. Eingaben mit $x \neq y$ müssen mit Wahrscheinlichkeit mindestens $1 - \frac{1}{n}$ verworfen werden.

   Hinweis: Fingerprinting.

2.2.2 Universelles Hashing

Offensichtlich können wir für jede Hashfunktion, ob für Hashing mit Verkettung oder für Hashing mit offener Addressierung, eine worst-case Laufzeit von $\Theta(n)$ erzwingen. Wir müssen also Glück haben, dass unsere Operationen kein worst-case Verhalten zeigen.

Aber was passiert, wenn wir mit einer Klasse H von Hashfunktionen arbeiten, anstatt mit einer einzelnen Hashfunktion? Zu Beginn der Berechnung wählen wir **zufällig** eine Hashfunktion $h \in H$ und führen Hashing mit Verkettung mit dieser Hashfunktion durch. Die Idee dieses Vorgehens ist, dass eine einzelne Hashfunktion durch eine bestimmte Operationenfolge zum Scheitern verurteilt ist, dass aber die meisten Hashfunktion mit Bravour bestehen. Die erste Frage: Was ist eine geeignete Klasse H?

Definition 2.18 Eine Menge $H \subseteq \{h \mid h : U \to \{0, ..., m-1\}\}$ ist **c-universell**, falls für alle $x, y \in U$ mit $x \neq y$ gilt, dass

$$|\{h \in H \mid h(x) = h(y)\}| \leq c \cdot \frac{|H|}{m}$$

Wenn H c-universell ist, dann gilt

$$\frac{|\{h \in H \mid h(x) = h(y)\}|}{|H|} \leq \frac{c}{m}$$

und es gibt somit keine zwei Schlüssel, die mit Wahrscheinlichkeit größer als $\frac{c}{m}$ auf die gleiche Zelle abgebildet werden. Gibt es c-universelle Klassen von Hashfunktionen für kleine Werte von c? Offensichtlich, man nehme zum Beispiel alle Hashfunktionen und wir erhalten $c = 1$. Diese Idee ist alles andere als genial: Wie wollen wir eine zufällig gewählte, **beliebige** Hashfunktion auswerten? Der folgende Satz liefert eine kleine Klasse von leicht auswertbaren Hashfunktionen.

Satz 2.19 *Es sei* $U = \{0, 1, 2, ..., p-1\}$ *für eine Primzahl* p. *Dann ist*

$$H = \{h_{a,b} \mid 0 \leq a, b < p, \ \ h_{a,b}(x) = ((ax + b) \bmod p) \bmod m\}$$

c-universell mit $c = (\lceil \frac{p}{m} \rceil \ / \ \frac{p}{m})^2$.

Die Hashfunktionen in H folgen dem Motto

erst durchschütteln ($x \mapsto y = (ax + b) \bmod p$) und dann hashen ($y \mapsto y \bmod m$)".

Beweis: Seien $x, y \in U$ beliebig gewählt. Es gelte $h_{a,b}(x) = h_{a,b}(y)$.
Dies ist genau dann der Fall, wenn es $q \in \{0, ..., m-1\}$ und $r, s \in \{0, ..., \lceil \frac{p}{m} \rceil - 1\}$ gibt mit

$$(*) \qquad \begin{array}{rcrcrcrcl} ax & + & b & = & q & + & r \cdot m & \bmod & p \\ ay & + & b & = & q & + & s \cdot m & \bmod & p. \end{array}$$

Für fixierte Werte von r, s und q ist dieses Gleichungssystem (mit den Unbekannten a und b) eindeutig lösbar. (Da p eine Primzahl ist, müssen wir ein Gleichungssystem über dem Körper Z_p lösen. Die Matrix des Gleichungssystem ist

$$\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix}$$

und damit regulär, denn $x \neq y$). Die Abbildung, die jedem Vektor (a, b) (mit $h_{a,b}(x) = h_{a,b}(y)$), den Vektor (q, r, s) mit Eigenschaft $(*)$ zuweist, ist somit bijektiv. Wir beachten, dass es $m \cdot \left(\lceil \frac{p}{m} \rceil\right)^2$ viele Vektoren (q, r, s) gibt und erhalten deshalb

$$\begin{aligned} |\{h \in H \mid h(x) = h(y)\}| &\leq m \cdot \left(\lceil \frac{p}{m} \rceil\right)^2 \\ &= \left(\lceil \frac{p}{m} \rceil / \frac{p}{m}\right)^2 \frac{p^2}{m} = c \cdot \frac{|H|}{m} \end{aligned}$$

denn $c = (\lceil \frac{p}{m} \rceil \ / \ \frac{p}{m})^2$ und $|H| = p^2$. □

Die Annahme „$U = \{0, ..., p-1\}$ für eine Primzahl p“ sieht auf den ersten Blick realitätsfremd aus, aber wer hindert uns daran U so zu vergrößern, dass die Mächtigkeit von der Form “Primzahl” ist? Niemand!

Sei H eine beliebige c-universelle Klasse von Hashfunktionen und sei eine *beliebige* Folge von $n-1$ `insert`-, `remove`- und `lookup`-Operationen vorgegeben. Wir betrachten die n-te Operation mit Operand x und möchten die erwartete Laufzeit dieser Operation bestimmen.

Beachte, dass der Erwartungswert über alle Hashfunktionen in H zu bestimmen ist. Die Operationenfolge ist fest vorgegeben und könnte möglicherweise von einem cleveren Gegner gewählt worden sein, um unser probabilistisches Hashingverfahren in die Knie zu zwingen. Der Gegner kennt sehr wohl die Klasse H, aber nicht die zufällig gewählte Funktion $h \in H$.

Sei S die Menge der vor Ausführung der n-ten Operation präsenten Elemente aus U. Die erwartete Laufzeit der n-ten Operation bezeichnen wir mit E_n. Die erwartete Anzahl von Elementen, mit denen der Operand x der n-ten Operation kollidiert, sei mit K_n bezeichnet. Dann gilt

$$\begin{aligned} E_n \;=\; 1 + K_n \;&=\; 1 + \frac{1}{|H|} \sum_{h \in H} |\{y \in S \mid h(x) = h(y)\}| \\ &=\; 1 + \frac{1}{|H|} \sum_{h \in H} \sum_{y \in S, h(x)=h(y)} 1 \\ &=\; 1 + \frac{1}{|H|} \sum_{y \in S} \sum_{h \in H, h(x)=h(y)} 1 \\ &\leq\; 1 + \frac{1}{|H|} \sum_{y \in S} c \cdot \frac{|H|}{m}, \end{aligned}$$

denn die Klasse H ist c-universell! Und damit folgt

$$\begin{aligned} E_n \;&\leq\; 1 + c \cdot \frac{|S|}{m} \\ &\leq\; 1 + c \cdot \frac{n-1}{m}. \end{aligned}$$

Jetzt können wir auch sofort die erwartete Laufzeit E der ersten n Operationen bestimmen:

$$\begin{aligned} E \;&=\; E_1 + ... + E_n \\ &\leq\; \sum_{i=1}^{n} \left(1 + c \cdot \frac{i-1}{m}\right) \\ &=\; n + \frac{c}{m} \cdot \sum_{i=1}^{n} (i-1) \\ &=\; n + \frac{c}{m} \cdot \frac{n \cdot (n-1)}{2} \\ &\leq\; n \cdot (1 + \frac{c}{2} \cdot \frac{n}{m}) \end{aligned}$$

und die erwartete Laufzeit ist linear, wenn $n = O\ (m)$.

Satz 2.20 *Sei H eine c-universelle Klasse von Hashfunktionen. Eine Folge von n Operationen sei beliebig, zum Beispiel in worst-case Manier, gewählt. Dann ist die erwartete Laufzeit aller n Operationen durch*

$$n\left(1+\frac{c}{2}\cdot\frac{n}{m}\right)$$

nach oben beschränkt.

Aufgabe 23
Sei $S \subseteq U$ eine Schlüsselmenge mit $|S| = n$. H sei eine Klasse c-universeller Hashfunktionen für Tabellengröße m. $B_i(h) = h^{-1}(i) \cap S$ ist die Menge der von h auf i abgebildeten Schlüssel.

(a) Zeige, daß bei zufälliger Wahl der Hashfunktion h aus H gilt:

$$E[\sum_i (|B_i(h)|^2 - |B_i(h)|)] \leq \frac{cn(n-1)}{m}.$$

(b) Die Ungleichung von Markov besagt: $Prob(x \geq aE[x]) \leq 1/a$ für alle Zufallsvariablen x und alle $a > 0$.
Zeige:

$$Prob(\sum_i |B_i(h)|^2 \geq n + a\frac{cn(n-1)}{m}) \leq 1/a.$$

(c) Bestimme m, so daß eine zufällige Hashfunktion mit Wahrscheinlichkeit $1/2$ injektiv auf S ist.

2.3 Stichproben

Der Entwurf randomisierter Algorithmen

durch die Berechnung auf Stichproben mit nachfolgender Extrapolation

ist ein erfolgreiches Entwurfsprinzip. Hier beschäftigen wir uns mit Anwendungen im Closest Pair Problem und in Datenströmen. Wir lernen weitere Anwendungen im Kapitel über Markoff-Ketten kennen.

2.3.1 Das Closest Pair Problem

Aufgabe 24
Eine Quelle liefert uns fortlaufend paarweise verschiedene Datensätze. Wir können k Datensätze gespeichert halten. Von einem angelieferten Datensatz müssen wir sofort entscheiden, ob wir ihn speichern oder ihn unwiederbringlich ignorieren. Entscheiden wir uns ihn zu speichern und ist kein Platz im Speicher mehr frei, so müssen wir auch sofort entscheiden, welcher der gespeicherten k Datensätze gelöscht wird.

Wir streben dabei eine Gleichverteilung an. Nach n Datensätzen mit $n \geq k$ soll also jede k-elementige Teilmenge der Datensätze mit Wahrscheinlichkeit $\frac{1}{\binom{n}{k}}$ als Stichprobe auftreten. Dabei ist uns die Zahl der Datensätze n zur Laufzeit **nicht** bekannt.

Beschreibe einen Algorithmus zur Lösung des Problems. Konzentriere dich dabei vor allem auf den Nachweis der Gleichverteilungseigenschaft.

Aufgabe 25
Ein Parallelrechner mit n Prozessoren soll das Maximum von n Zahlen in erwartet konstanter Zeit bestimmen. Jeder Prozessor erhält eine Eingabezahl. Zur Verfügung steht eine Zähleinheit, die n Bits als Eingabe erhält und die Anzahl der Einsen in der Eingabe in konstanter Zeit bestimmt.

Darüber hinaus können wir eine Maximumseinheit benutzen. Der Maximumseinheit kann eine beliebig große Menge von Zahlen übergeben werden. Innerhalb von einem Schritt gibt sie das Maximum aller Eingabezahlen zurück. Allerdings entstehen uns für jede Berechnung eines Maximums aus k Zahlen Kosten in Höhe von k^2 Einheiten.

Beschreibe einen Las Vegas Algorithmus, der in erwartet konstanter Zeit das Maximum von n Zahlen bestimmt und dabei erwartete Kosten von $O(n)$ verursacht. Jedem Prozessor steht eine eigene Zufallsquelle zur Verfügung.

Im Closest-Pair Problem sind n Punkten $p_1, \ldots, p_n \in \mathbb{R}^2$ gegeben und ein Paar (p_i, p_j) nächstliegender Punkte ist zu bestimmen. Mit Hilfe von Voronoi-Diagrammen lässt sich das Closest-Pair Problem in Zeit $O(n \cdot \log_2 n)$ lösen. Wir stellen hier einen weitaus schnelleren Linearzeit-Algorithmus vor.

Der Minimalabstand zwischen zwei Punkten sei δ. Der Algorithmus berechnet ein Gitter Γ_μ mit quadratischen Zellen der Seitenlänge $\mu \geq \delta$. Wir machen eine Reihe einfacher Beobachtungen.

- Die Punkte eines nächstliegenden Paares liegen in derselben oder benachbarten Zellen des Gitters Γ_μ.
- Eine Zelle von Γ_μ enthält höchstens $(2 \cdot \frac{\mu}{\delta})^2$ Punkte.

Das zentrale Problem ist also die Berechnung einer guten Approximation des Minimalabstands δ. Unser Algorithmus wählt einen zufälligen Punkt $p \in P$ als Stichprobe und berechnet den minimalen Abstand $\delta(p)$ von p zu einem Punkt aus P. Wir machen Fortschritte in der Approximation von δ, denn im Erwartungsfall gilt $\delta(q) \geq \delta(p)$ für mindestens die Hälfte aller Punkte q.

Algorithmus 2.21

(1) Sei $P_0 = P$ die Menge der vorgegebenen Punkte. Setze $i = 0$.

(2) Wiederhole, solange bis $P_i = \emptyset$:

(2a) Wähle einen Punkt $p \in P_i$ zufällig. Bestimme $\delta_i = \min\{\ ||p - q|| \mid q \in P_i,\ p \neq q\ \}$.

Kommentar: Beachte, dass die Laufzeit proportional zur Größe von P_i ist.

(2b) Setze $Q = P_i$ und entferne alle Punkte aus Q, die keinen weiteren Punkt aus P_i in ihrer oder einer benachbarten Zelle von $\Gamma_{\delta_i/3}$ besitzen. Wir nennen die entfernten Punkte *isoliert*. Setze $P_{i+1} = Q$.

Kommentar: Es ist stets $\delta \leq \delta_i$. Wenn $P_{i+1} = \emptyset$ nach der Entfernung isolierter Punkte, dann ist offensichtlich $\frac{\delta_i}{3} \leq \delta \leq \delta_i$ und insbesondere ist $\delta \leq \delta_i \leq 3 \cdot \delta$. Wir haben in diesem Fall also eine gute Approximation von δ gefunden.

Achtung: Wie bestimmt man die Menge der isolierten Punkte in Linearzeit?

(2c) Setze $i = i + 1$.

(3) Setze $i = i - 1$. Bestimme den Abstand zwischen je zwei Punkten $p, q \in P_i$, die sich in derselben oder benachbarten Zellen von Γ_{δ_i} befinden. Ermittle den Minimalabstand.

Kommentar: Es ist $\delta \leq \delta_i \leq 3 \cdot \delta$ und damit befinden sich höchstens $(2 \cdot 3)^2 = 36$ Punkte in einer Zelle von Γ_{δ_i}. Schritt (3) verläuft also in linearer Zeit.

Bis auf die Implementierung von Schritt (2b) ist der Algorithmus vollständig beschrieben.

Aufgabe 26
Implementiere Schritt (2b) in erwarteter Zeit $O(|P_i|)$.
Hinweis: Hashing.

Satz 2.22 *Algorithmus 2.21 bestimmt den Minimalabstand einer Punktemenge im $\mathbb{R}^2$ in erwartet linearer Zeit.*

Beweis: Da Schritt (3) sogar in worst-case linearer Zeit verläuft, genügt der Nachweis von

$$\sum_{i=1}^{\infty} E[\ |P_i|\ ] = O(n). \tag{2.2}$$

Was ist die erwartete Größe von P_i?

Behauptung 2.1 *Wenn Schritt (2) für eine Menge P_i ausgeführt wird, dann werden im Erwartungsfall mindestens die Hälfte aller Punkte entfernt.*

Beweis der Behauptung: Wir definieren $\delta(p) = \min\{\ ||p-q|| \mid q \in P_i,\ p \neq q\ \}$ als den Abstand von $p \in P_i$ zu einem nächstliegenden Punkt in P_i. Wenn $\mu \leq \delta(p)$, dann ist p im Gitter $\Gamma_{\mu/3}$ isoliert, denn der Abstand zwischen Punkten aus benachbarten Zellen von $\Gamma_{\mu/3}$ ist höchstens $\sqrt{8} \cdot \mu/3 < \mu$.

Für $P_i = \{q_1, \ldots q_m\}$ gelte o.B.d.A. $\delta(q_1) \leq \cdots \leq \delta(q_m)$. Nach der obigen Überlegung sind also mindestens $m-k+1$ Punkte aus P_i im Gitter $\Gamma_{\delta(q_k)/3}$ isoliert. Wählen wir einen Punkt zufällig aus der Menge P_i, dann ist die erwartetet Anzahl isolierter Punkte damit mindestens

$$\sum_{k=1}^{m} \frac{m-k+1}{m} = \frac{m \cdot (m+1)}{2 \cdot m} = \frac{m+1}{2}$$

und das war zu zeigen.

Zum Nachweis von (2.2) genügt der Nachweis von

$$E[\ |P_i|\ ] \leq \frac{n}{2^i}.$$

Wir führen einen induktiven Beweis, der im Basisfall wegen $P_0 = P$ trivial ist. Für den induktiven Schritt beachten wir

$$\begin{aligned} E[\ |P_{i+1}|\ ] &= \sum_{m=0}^{\infty} \text{prob}[\ |P_i| = m\ ] \cdot E[\ |P_{i+1}| \mid |P_i| = m\ ] \\ &\leq \sum_{m=0}^{\infty} \text{prob}[\ |P_i| = m\ ] \cdot \frac{m-1}{2} \\ &\leq \frac{1}{2} E[\ |P_i|\ ] \leq \frac{n}{2^{i+1}}, \end{aligned}$$

wobei die letzte Ungleichung aus der Induktionshypothese folgt. □

2.3.2 Zählen in Datenströmen

Im *Datenstrom Modell* (engl. streaming data model) strömen Daten fortlaufend ein und Berechnungen sind in Echtzeit, bzw in wenigen Datendurchläufen zu erbringen. Beispiele sind die fortlaufende Protokollierung von Telefonanrufen durch weltweit agierende Telefonunternehmen und die damit verbundenen Reaktionen auf überlastete Leitungen oder die Datenanalyse in der Abwehr einer Denial-of-Service Attacke.

Aufgabe 27
Wir erhalten einen Datenstrom von $n-1$ verschiedenen Zahlen aus der Menge $\{1,\ldots,n\}$ und möchten feststellen welche Zahl fehlt. Wir dürfen mit einem Speicher der Größe $O(\log_2 n)$ arbeiten. Entwirf einen Algorithmus, der die Zahlen der Reihe nach **einmal** liest und dann die fehlende Zahl bestimmt.

Wir nehmen an, dass $(x_n \mid n \in \mathbb{N})$ der Datenstrom ist, wobei der Schlüssel x_n zum Zeitpunkt n erscheine. Wir möchten für jeden Zeitpunkt n zumindest approximativ feststellen, wieviele *verschiedene* Schlüssel wir bisher erhalten haben. Das Problem ist nicht-trivial, denn wir müssen annehmen, dass das Universum aller möglichen Schlüssel extrem groß ist und eine Abspeicherung der bisher erhaltenen verschiedenen Schlüssel für große Werte von n deshalb nicht möglich ist.

Wir versuchen stattdessen, eine verlässliche *Stichprobe verschiedener Schlüssel* zu berechnen. Dazu weisen wir jedem gesehenen Schlüssel eine zufällig berechnete Priorität zu und halten alle Schlüssel ab einer Mindestpriorität als Stichprobe fest. Wenn die Stichprobe überläuft, dann ist die Mindestpriorität um 1 hochzusetzen und alle Schlüssel mit zu kleiner Priorität werden entfernt.

Algorithmus 2.23 **Bestimmung einer Stichprobe verschiedener Schlüssel**

(0) m sei eine Zweierpotenz.

(1) Bestimmung der Prioritäten. Wähle Parameter $A \in \{1,\ldots,m-1\}$ und $B \in \{0,\ldots,m-1\}$ zufällig und definiere die Hashfunktion $h_{A,B}(u) = (A{\cdot}u{+}B) \bmod m$. Schließlich wähle

$$p(u) = \text{ die Anzahl der führenden Nullen in der Binärdarstellung von } h_{A,B}(u)$$

als Prioritätszuweisung. (Wir fordern, dass die Binärdarstellung die exakte Länge $\log_2 m$ besitzt.)

Kommentar: Beachte, dass $\text{prob}[\, p(u) \geq k \,] = 2^{-k}$.

(2) Setze PRIORITÄT$= 0$ und STICHPROBE$= \emptyset$. S sei die erlaubte Maximalgröße einer Stichprobe.

Wiederhole für $i = 1,\ldots,n$:

(2a) Füge x_i zur Menge STICHPROBE hinzu, falls $p(x_i) \geq$ PRIORITÄT.

(2b) Solange STICHPROBE mehr als S Schlüssel besitzt, entferne alle Schlüssel x mit $p(x) =$ PRIORITÄT und erhöhe PRIORITÄT um 1.

(3) Setze $p =$PRIORITÄT und gib $2^p \cdot |\text{STICHPROBE}|$ als Schätzung aus.

Es ist $\text{prob}[\, p(x) \geq p \,] = 2^{-p}$ und deshalb folgt für $p =$ PRIORITÄT

$$\begin{aligned} E[\, |\text{STICHPROBE}| \,] &= \sum_{x \text{ ein Schlüssel}} \text{prob}[\, p(x) \geq p \,] \\ &= 2^{-p} \cdot \text{Anzahl verschiedener Schlüssel.} \end{aligned}$$

Die Schätzung in Schritt (3) ist also gut, wenn große Abweichungen vom Erwartungswert $E[\, |\text{STICHPROBE}| \,]$ nicht wahrscheinlich sind.

Aufgabe 28
Bestimme die Varianz $V[\ |\text{STICHPROBE}|\ ]$ und zeige, dass $V[\ |\text{STICHPROBE}|\ ] = O(E[\ |\text{STICHPROBE}|\ ])$.

Wir setzen $E = E[\ |\text{STICHPROBE}|\ ]$ und die Tschebyscheff Ungleichung liefert somit

$$\text{prob}[\ |\ |\text{STICHPROBE}| - E| > \varepsilon \cdot E\ ] = O(\frac{E}{\varepsilon^2 \cdot E^2}) = O(\frac{1}{\varepsilon^2 \cdot E}).$$

Sei V die Anzahl der verschiedenen Schlüssel. Dann ist $V = 2^p \cdot E$ und wir erhalten

$$\begin{aligned} & \text{prob}[\ |\ 2^p \cdot |\text{STICHPROBE}| - V| > \varepsilon \cdot V\ ] \\ = \ & \text{prob}[\ |\ 2^p \cdot |\text{STICHPROBE}| - 2^p \cdot E| > \varepsilon \cdot 2^p \cdot E\ ] \\ = \ & O(\frac{1}{\varepsilon^2 \cdot E}) \end{aligned}$$

Unsere Schätzung wird also umso exakter sein, je größer der Erwartungswert E ist, d.h. je größer die tatsächliche Anzahl verschiedener Schlüssel ist. Abweichungen um $\sqrt{E}$ sind allerdings durchaus wahrscheinlich.

Wie können wir die Verläßlichkeit „boosten“, also mit mehreren Messungen eine exaktere Schätzung erhalten? Wir kennen das richtige Vorgehen bereits aus Beispiel 1.4, wir sollten nämlich den Median aller Messungen als unsere Schätzung ausgeben:

Aufgabe 29
Es gelte $\text{prob}[\ |\ 2^p \cdot |\text{STICHPROBE}| - V| > \varepsilon \cdot V\ ] < \frac{1}{4}$.

Wir setzen $S = \frac{3}{\varepsilon^2}$ und wiederholen Algorithmus 2.23 $O(\ln(\frac{1}{\delta}))$-mal mit zufälligen und unabhängig voneinander gewählten Prioritätszuweisungen. Wir bestimmen alle Stichprobengrößen (nach entsprechender Skalierung mit 2^p) und geben den Median aus.

Zeige: Wir erhalten eine bis auf den Faktor $1 + \varepsilon$ exakte Schätzung mit Wahrscheinlichkeit $1 - \delta$. Also genügt die Speicherplatzkomplexität $O(\frac{1}{\varepsilon^2} \cdot \ln(\frac{1}{\delta}))$.

2.4 Randomisierung in Konfliktsituationen

Wir betrachten das **Byzantine Agreement** Problem: Die Armee von Byzanz ist in mehrere Divisionen aufgeteilt. Jede Division wird von einem loyalen oder illoyalen General befehligt. Die Generäle kommunizieren über Boten, um sich entweder auf „Angriff“ oder auf „Rückzug“ einheitlich festzulegen; die illoyalen Generäle versuchen allerdings durch betrügerische Nachrichten eine Einigung auf eine sinnvolle Entscheidung zu verhindern. Kann trotzdem ein sinnvolles einheitliches Vorgehen der loyalen Generäle erreicht werden? Insbesondere ist zu fordern, dass eine bestimmte Vorgehensweise dann umzusetzen ist, wenn alle loyalen Generäle aufgrund ihrer individuellen Beobachtungen diese Vorgehensweise vorschlagen.

In der Sprache verteilter Systeme erhalten wir die folgende Formalisierung: Ein Netzwerk von n Prozessoren ist vorgegeben, wobei t Prozessoren fehlerhaft sind und möglicherweise versuchen, die Berechnung des Netzwerks bösartig zu stören. Jeder der Prozessoren kann mit jedem anderen Prozessor in einem Berechnungsschritt „reden“. Zu Anfang starten die Prozessoren mit jeweils einem Eingabebit. Wir möchten ein Protokoll mit den folgenden Eigenschaften herleiten:

(a) Die fehlerfreien Prozessoren haben sich am Ende der Berechnung auf ein Bit b geeinigt.

(b) Wenn alle fehlerfreien Prozessoren dasselbe Eingabebit c besitzen, dann ist eine Einigung auf Bit c erforderlich.

Natürlich können die fehlerfreien Prozessoren Eingabebits ihrer Kollegen anfordern und eine Mehrheitsentscheidung durchführen. Dieses Vorgehen kann aber durch bösartige Prozessoren hintertrieben werden, wenn Mehrheiten für unterschiedliche fehlerfreie Prozessoren durch unterschiedliche Kommunikationen gekippt werden.

Im folgenden nehmen wir an, dass $t < \frac{n}{8}$, und wir erlauben somit eine beachtliche Zahl bösartiger Prozessoren. Desweiteren fordern wir, dass in jedem Berechnungsschritt ein globales Zufallsbit zur Verfügung steht. Das folgende Protokoll erreicht eine Einigung in erwarteter konstanter Zeit und ist resistent gegen jede Form von Bösartigkeit.

Algorithmus 2.24 **Byzantine Agreement**

(1) Setze $T = \frac{n}{8}$. Die Schwellenwerte NIEDRIG= $\frac{n}{2} + T + 1$, Hoch= $\frac{n}{2} + 2T + 1$ und ENTSCHEIDUNG= $\frac{n}{2} + 3T + 1$ werden benutzt.

Kommentar: Wir beschreiben im Folgenden nur die Aktionen der fehlerfreien Prozessoren. Beachte, dass ENTSCHEIDUNG $= \frac{n}{2} + 3T + 1 = \frac{7n}{8} + 1$ gilt.

(2) Prozessor i besitze b_i als sein Eingabebit und votiert anfänglich für b_i.

(3) Wiederhole, solange bis eine Entscheidung getroffen wird:

(3a) Jeder Prozessor sendet sein Votum an alle anderen Prozessoren.

Kommentar: Bösartige Prozessoren kommunizieren möglicherweise verschiedene Voten oder verschicken keine Voten. Bei Nichterhalt eines Votums wird das Bit 0 als „empfangen“ angesehen.

(3b) Jeder Prozessor empfängt die Voten seiner Kollegen. Erhält Prozessor i eine Mehrheit von Voten für Bit 1, dann wird $\text{Bit}_i = 1$ und ansonsten $\text{Bit}_i = 0$ gesetzt. Stimmen_i ist die Stimmenanzahl für das gewinnende Bit.

(3c) Die Prozessoren wählen den Schwellenwert $S \in \{$ NIEDRIG, HOCH $\}$, abhängig vom Wert des globalen Zufallsbits.

(3d) Wenn $\text{Stimmen}_i \geq S$, dann behält Prozessor i seine bisherige Wahl bei und setzt ansonsten $\text{Bit}_i = 0$.

(3e) Wenn $\text{Stimmen}_i \geq$ ENTSCHEIDUNG, dann legt sich Prozessor i endgültig auf seine bisherige Wahl fest.

Für die Analyse des Protokolls ist die folgende einfache Beobachtung wesentlich.

In jeder Runde gilt für je zwei fehlerfreie Prozessoren, dass die erhaltenen Stimmen für Bit 0 bzw. Bit 1 um höchstens t differieren, denn nur bösartige Prozessoren können unterschiedliche Voten abgeben.

Als Konsequenz erhalten wir die folgenden Beobachtungen.

(1) Wenn alle fehlerfreien Prozessoren eine Runde mit demselben Bit beginnen, dann entscheiden sie sich alle in dieser Runde, denn ihre Stimmenmehrheit ist mindestens $n - t \geq \frac{7n}{8} + 1 \geq$ ENTSCHEIDUNG. Insbesondere ist also Eigenschaft (b) des Byzantine Agreement erfüllt.

(2) Wenn für *irgendeinen* fehlerfreien Prozessor Stimmen$_i \geq$ HOCH $= \frac{n}{2} + 2T + 1$, dann übertrifft die Stimmenmehrheit eines jeden fehlerfreien Prozessors den Schwellenwert NIEDRIG. Mit Wahrscheinlichkeit $\frac{1}{2}$ wird aber NIEDRIG als Schwellenwert in Schritt (3c) gewählt und alle fehlerfreien Prozessoren beginnen in diesem Fall die nächste Runde mit demselben Bit.

Wir wenden Beobachtung (1) an und erhalten, dass unter dieser Fallannahme eine Festlegung mit Wahrscheinlichkeit $\frac{1}{2}$ in der nächsten Runde erfolgt.

(3) Wenn für *alle* fehlerfreien Prozessor Stimmen$_i <$ HOCH $= \frac{n}{2} + 2T + 1$, dann setzen alle fehlerfreien Prozessoren mit Wahrscheinlichkeit $\frac{1}{2}$ ihr Bit in Schritt (3d) auf Null, da der Schwellenwert HOCH mit Wahrscheinlichkeit $\frac{1}{2}$ in Schritt (3c) gewählt wird.

Also erfolgt auch unter dieser Fallannahme eine Festlegung mit Wahrscheinlichkeit $\frac{1}{2}$ in der nächsten Runde.

Die Beobachtungen (2) und (3) zeigen, dass mit Wahrscheinlichkeit $\frac{1}{2}$ eine Festlegung in der nächsten Runde erfolgt.

Satz 2.25 *Algorithmus 2.24 erreicht eine Festlegung nach einer erwarteten Zahl von höchstens drei Runden.*

Fazit: Algorithmus 2.24 randomisiert im Schritt (3c) erfolgreich *gegen* die Aktionen bösartiger Prozessoren. Ein ähnliches Vorgehen ist auch für on-line Algorithmen erfolgreich, wenn *gegen die Zukunft* randomisiert wird. In beiden Fällen stellt sich heraus, dass sich ein Gegner (also bösartige Prozessoren oder zukünftige Eingaben) nicht gegen die verschiedenen Aktionen nach Randomisierung durchsetzen kann: Die Aktionen randomisierter Algorithmen sind nicht voraussagbar.

Aufgabe 30
Zeige: Wenn sich ein fehlerfreier Prozessor festgelegt hat, dann werden sich alle weiteren fehlerfreien Prozessoren spätestens in der nächsten Runde festlegen: Prozessoren müssen deshalb im *worst-case* höchstens eine Runde warten.

Aufgabe 31
Wir haben gezeigt, dass weniger als $\frac{n}{8}$ fehlerhafte Prozessoren die fehlerfreien Prozessoren nicht an einer Einigung innerhalb konstanter erwarteter Rundenzahl hindern können. Zeige, dass durch geschickte Wahl der Schwellenwerte auch eine Anzahl von αn fehlerhaften Prozessoren mit $\alpha > \frac{1}{8}$ toleriert werden kann.

Aufgabe 32
Wir betrachten Algorithmus 2.24. Wir nehmen nun in einem neuen Modell an, dass alle Zufallsbits allen Prozessoren *von vornherein* bekannt sind.

Zeige, dass es Eingaben für die fehlerfreien Prozessoren gibt, so dass $\frac{n}{8} - 1$ fehlerhaften Prozessoren eine Einigung verhindern können.

2.5 Symmetry Breaking

Wir betrachten parallele Graph-Algorithmen für Shared Memory Architekturen: Parallel arbeitende Prozessoren können auf die Register eines gemeinsamen Speichers entweder mit Lese- oder Schreiboperationen zugreifen. Wir erlauben, dass jedes Register von mehreren Prozessoren lesbar ist. Sollten mehrere Prozessoren dasselbe Register beschreiben wollen, so "gewinnt" irgendeiner der beteiligten Prozessoren, das parallele Programm weiss aber nicht welcher.

In effizienten parallelen Algorithmen werden Prozessoren die überwiegende Zeit auf den ihnen zugewiesenen Aufgaben arbeiten und die teure Kommunikation mit anderen Prozessoren auf ein absolutes Minimum reduzieren. Wie erreicht man es dann ohne Kommunikation,

dass sich die Prozessoren in verschiedene Typen mit unterschiedlichen Zielen aufspalten? Die Methode des Brechens der Symmetrie stellt sich hier als sehr erfolgreich heraus.

2.5.1 Zusammenhangskomponenten

Sei $G = (V, E)$ ein ungerichteter Graph mit n Knoten und m Kanten. Unser Ziel ist die Bestimmung aller Zusammenhangskomponenten[3].

Unser Algorithmus beginnt mit einem Wald aus Einzelknoten, wobei jeder Knoten über eine Eigenchleife mit sich selbst verbunden ist. Knoten mit Eigenschleifen sind ein triviales Beispiel eines Sterns: Ein Stern besteht aus einem ausgezeichneten Knoten, der Wurzel und anderen Knoten, die sämtlich eine Kante zur Wurzel besitzen.

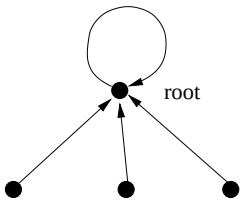


Abbildung 2.1: Ein Stern.

Sterne werden stets Teilmengen von Zusammenhangskomponenten sein. Wenn zwei Sterne Teilmengen derselben Zusammenhangskomponente sind, dann können wir sie verschmelzen: Wir wählen eine Wurzel aus, verbinden alle Knoten des anderen Sterns mit der ausgewählten Wurzel und erhalten einen Stern für die Vereinigungsmenge.

Welche Sterne sollen wir miteinander verschmelzen? Wir müssen auf jeden Fall eine sequentielle Kette von Verschmelzungsschritten (mache Stern s_2 zu einem Teilstern von s_1, s_3 zu einem Teilstern von s_2, s_4 zu einem Teilstern von s_3 etc.) vermeiden, wenn wir eine schnelle Berechnung anstreben.

Algorithmus 2.26 **Die Bestimmung von Zusammenhangskomponenten**
/* Der ungerichtete Graph $G = (V, E)$ mit $|V| = n$ und $|E| = m$ ist gegeben. Wir arbeiten mit $n + m$ Processoren, nämlich einem Prozessor für jeden Knoten und jede Kante von G. */

(1) for $i = 1$ to n pardo Vater$[i] = i$;

/* Das Vater-Array definiert n Sterne aus Einzelknoten. Eine Kante $\{u, v\} \in E$ heißt lebendig, wenn u und v zu verschiedenen Sternen gehören. Beachte, dass zu Anfang alle Kanten lebendig sind. */

(2) while (G hat lebendige Kanten) do

(a) Jede Wurzel w, also jeder Knoten w mit Vater$[w] = w$, wählt ein Geschlecht $g(w) \in \{0, 1\}$ *zufällig* und weist sein Geschlecht den Kindern zu. Wir interpretieren 0 als männlich und 1 als weiblich.

/* Wir brechen die Symmetrie durch Zufallsentscheidungen. */

[3]Eine Teilmenge $C \subseteq V$ ist genau dann eine Zusammenhangskomponente von G, wenn je zwei Knoten von C durch einen Weg verbunden sind und wenn es keinen Knoten in $V \setminus C$ gibt, der mit einem Knoten in C durch einen Weg verbunden ist.

(b) Angenommen, u gehört zu einem männlichen Stern m und v zu einem weiblichen Stern w. Der für die Kante $\{u, v\}$ verantwortliche Prozessor versucht, w in das Register von m zu schreiben.

/* Irgendein Kanten-Prozessor wird gewinnen, wir wissen aber nicht welcher. */

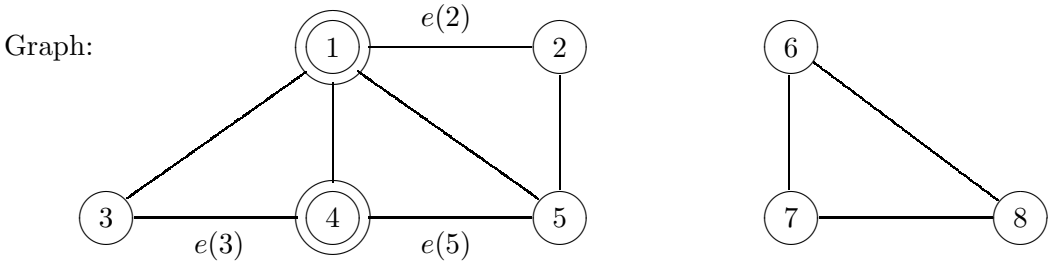
(c) Wenn $w(m)$ der Inhalt des Registers von m ist, dann wird die Wurzel von m durch eine Kante mit der Wurzel von $w(m)$ verbunden: m wird zu einem Teilstern von $w(m)$.

/* Wir haben Ketten von Verschmelzungsschritten verhindert! */

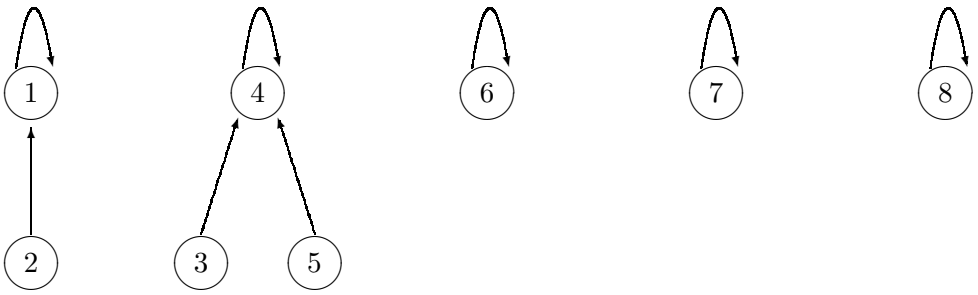
(d) for $i = 1$ to n pardo Vater[i] = Vater[Vater[i]];

/* Die Stern-Eigenschaft ist wieder hergestellt. */

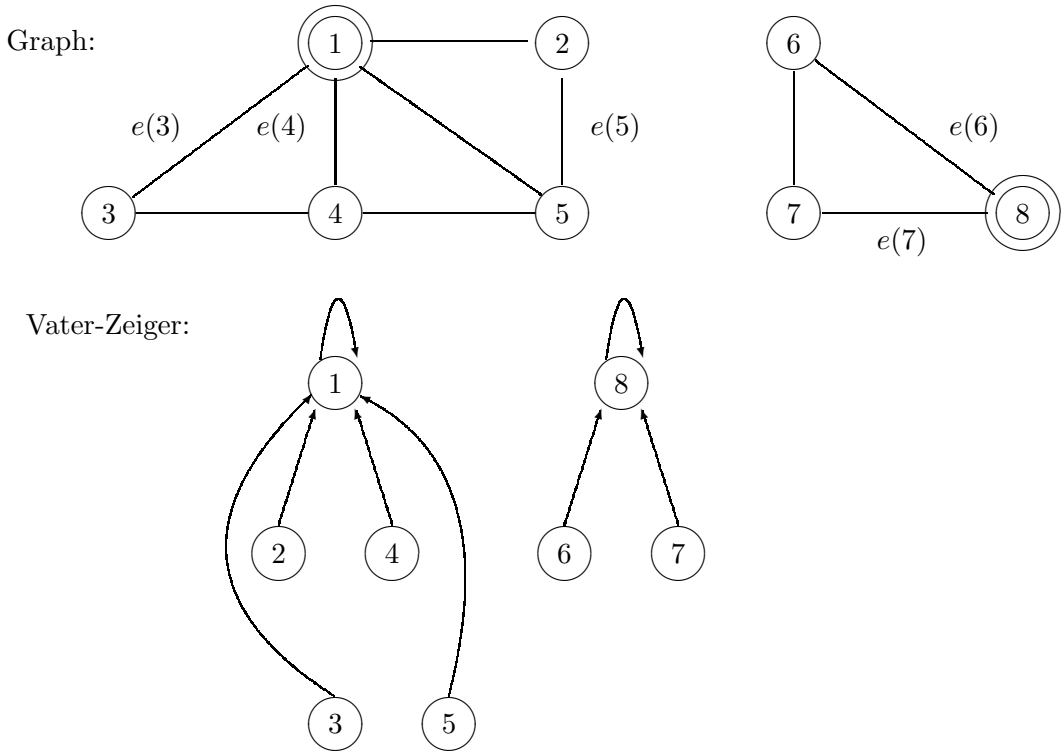
Beispiel 2.3 Wir zeigen das Ergebnis der ersten Iteration für einen Graph von 8 Knoten. Weibliche Knoten besitzen zwei Kreise und die von männlichen Knoten ausgewählten Kanten sind markiert.



Vater-Zeiger:



Wir haben fünf Sterne erhalten. Für die zweite Iteration nehmen wir an, dass die Sterne der Knoten 1 und 8 weiblich sind und dass alle anderen Sterne männlich sind.



$\{1, 2, 3, 4, 5\}$ und $\{6, 7, 8\}$ sind die Zusammenhangskomponenten von G.

Wir beginnen unsere Analyse mit der folgenden Beobachtung.

Lemma 2.27

(a) Zu jeder Zeit gehören alle Knoten eines Sterns derselben Zusammenhangskomponente an.

(b) Algorithmus 2.26 berechnet alle Zusammenhangskomponenten.

(c) Jede Iteration der while-Schleife läuft in Zeit $O(1)$.

Wir sagen, dass ein Stern lebendig ist, wenn einer seiner Knoten Endpunkt einer lebendigen Kante ist. Wir sagen, dass die Wurzel eines Sterns verschwindet, wenn der Stern an einen anderen Stern angehängt wird.

Lemma 2.28 *S sei ein Stern. Wenn S eine echte Teilmenge einer Zusammenhangskomponente ist, dann verschwindet S in einer Iteration mit Wahrscheinlichkeit mindestens $\frac{1}{4}$.*

Proof: Betrachte eine lebendige Kante $\{u, v\}$ für einen Knoten $u \in S$. Mit Wahrscheinlichkeit $\frac{1}{4}$ wird der Stern von u männlich und der Stern von v weiblich. Die Wurzel von S verschwindet also mit Wahrscheinlichkeit mindestens $\frac{1}{4}$. □

Satz 2.29 *Die erwartete Laufzeit von Algorithmus 2.26 ist durch $O(\log_2 n)$ beschränkt. Insbesondere genügen $5 \cdot \log_2 n$ Iterationen der while-Schleife mit Wahrscheinlichkeit mindestens $1 - \frac{1}{n}$. Wir arbeiten mit $n + m$ Prozessoren.*

Beweis: Wir wählen einen beliebigen Knoten w aus und bestimmen die Wahrscheinlichkeit p_w, dass w nicht nach $5 \cdot \log_2 n$ Iterationen verschwindet:

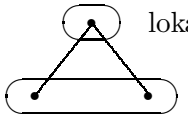
$$p_w \leq \left(1 - \frac{1}{4}\right)^{5 \cdot \log_2 n} = \left(\frac{3}{4}\right)^{5 \cdot \log_2 n} = \left(\frac{243}{1024}\right)^{\log_2 n} \leq \left(\frac{1}{4}\right)^{\log_2 n} = 2^{-2 \cdot \log_2 n} = n^{-2}.$$

Damit ist also die Wahrscheinlichkeit, dass irgeindeiner der n Knoten nicht nach $5 \cdot \log_2 n$ Iterationen verschwindet, höchstens $n \cdot n^{-2} = \frac{1}{n}$ und das war zu zeigen. Die nächte Übungsaufgabe behandelt die Analyse der erwarteten Laufzeit. □

Aufgabe 33
Zeige, dass Algorithmus 2.26 die erwartete Laufzeit höchstens $O(\log_2 n)$ besitzt.

2.5.2 Maximale unabhängige Mengen

Für einen ungerichteten Graphen $G = (V, E)$ suchen wir nach einer größten Menge $I \subseteq V$, so dass keine zwei Knoten benachbart sind. Das Independent Set Problem ist NP-vollständig, und wir sind deshalb sogar zufrieden, wenn wir eine lokal maximale, also eine nicht-vergrößerbare unabhängige Menge I bestimmen können. Das folgende Beispiel zeigt die größte unabhängige Menge und ein lokales Maximum für einen Graphen mit drei Knoten.



Hier ist ein trivialer Algorithmus für die Bestimmung einer lokal maximalen unabhängigen Menge.

Algorithmus 2.30 **Der sequentielle Independent Set Algorithmus.**

(1) Der ungerichtete Graph $G = (V, E)$ ist gegeben. Setze $I = \emptyset$.

(2) Solange $V \neq \emptyset$ wiederhole

> Entferne einen beliebigen Knoten v und alle Nachbarn von v aus der Menge V. Füge v zu I hinzu.

Können wir die WHILE-Schleife in Schritt (2) parallelisieren oder ist die Bestimmung einer nicht vergrößerbaren unabhängigen Menge ein Problem, das nur inhärent sequentielle Lösungen erlaubt? Wir beschreiben im Folgenden eine superschnelle Lösung, nämlich einen Algorithmus, der in logarithmischer Zeit auf einem Parallelrechner arbeitet. Wir definieren

$$N(u) := \{v \in V \mid \{v, u\} \in E\}$$

als die Menge der Nachbarn von u und erweitern diesen Begriff auf Knotenmengen $X \subseteq V$ durch

$$N(X) := \bigcup_{u \in X} N(u).$$

$d(u) = \mid N(u) \mid$ ist der Grad von Knoten u.

Algorithmus 2.31 **Der parallele Independent Set Algorithmus.**

(1) Setze $I = \emptyset$.

(2) WHILE $V \neq \emptyset$ DO

(2a) FOR $v \in V$ PARDO

Wenn v isoliert ist, dann füge v zu I hinzu und entferne v aus V.
Ansonsten wird v mit Wahrscheinlichkeit $\frac{1}{2 \cdot d(v)}$ markiert.
/* Wir versuchen in einem Schritt möglichst viele markierte Knoten zu I hinzuzufügen. Problematisch ist der Fall, dass eine Kante zwei markierte Knoten verbindet. */

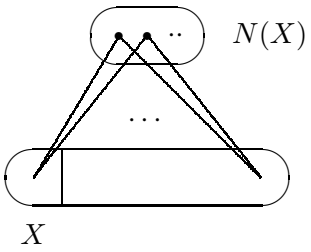
(2b) FOR $\{u, v\} \in E$ mit markierten Endpunkten u und v PARDO

Lösche die Markierung des Endpunkt mit dem kleineren Grad, bzw. lösche beide Markierungen bei Gleichheit der Grade.

(2c) Sei X die Menge markierter Knoten. Setze $I = I \cup X$ und $V = V \setminus \big(X \cup N(X)\big)$.
Kommentar: Die Menge I ist unabhängig, aber möglicherweise noch nicht maximal. Wir haben X und seine Nachbarn aus V entfernt, um die nächste Iteration vorzubereiten.

Kommentar: Während die While-Schleife sequentiell auszuführen ist, werden die Anweisungen (2a) und (2b) parallel für jeden Knoten $v \in V$, bzw. für jede Kante $\{u, v\} \in E$ ausgeführt.

Algorithmus 2.31 führt Symmetry Breaking durch die zufällige Vorauswahl in Verbindung mit der Grad-Regel durch. Zur Begründung dieser Kombination betrachten wir den vollständigen bipartiten Graphen mit k Knoten in der ersten Schicht und $n - k$ Knoten in der zweiten Schicht. k sei zuerst sehr viel kleiner als $n - k$ gewählt.



Wenn Algorithmus 2.31 nur die niedriggradigen Knoten, also die Knoten in X markiert, dann überleben alle markierten Knoten den „Kantentest“ in Schritt (2b) und alle hochgradigen Knoten werden in Schritt (2c) eliminiert. Der Algorithmus wird dann die unabhängige Menge der niedriggradigen Knoten im nächsten Schritt ausgeben. *Wenn* Algorithmus 2.31 hingegen mindestens einen hochgradigen Knoten markiert, dann gewinnen markierte hochgradige Knoten die Kantentests und die unabhängige Menge der hochgradigen Knoten wird im nächsten Schritt ausgegeben.

Die Vorauswahl kann zu einer drastischen Reduktion der Anzahl markierter Knoten führen. Wenn wir wieder den obigen vollständigen bipartiten Graphen, aber diesmal mit $k = \frac{n}{2} = n - k$ betrachten, dann wird ein Knoten nur mit Wahrscheinlichkeit $\frac{1}{n}$ markiert und X hat die erwartete Größe 1. Aber zumindest in diesem Beispiel ist das nicht weiter schlimm, denn wenn

X aus genau einem Knoten u besteht, dann wird u zur unabhängigen Menge hinzugenommen und eliminiert dann alle Nachbarn und damit alle Kanten.

Dieser Fortschritt durch Eliminierung vieler Kanten tritt stets auf, wie wir später sehen werden: Jede Iteration von Schritt (2) eliminiert einen konstanten Prozentsatz aller Kanten und deshalb sind nur logarithmisch viele Iterationen notwendig.

Wir betrachten im Folgenden eine beliebige Iteration von Schritt (2) und zeigen zuerst, dass ein markierter Knoten v mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ in die unabhängige Menge aufgenommen wird. Die Grad-Regel erweist sich hier als große Hilfe.

Lemma 2.32 *Für jeden markierten Knoten $v \in V$ gilt*

$$\text{prob}[\ v \in I\ ] \geq \frac{1}{2}.$$

Beweis: Ein markierter Knoten v wird genau dann eliminiert, wenn ein Knoten aus

$$\text{Gefährlich}(v) = \{\ u \in N(v) \mid d(u) \geq d(v)\}$$

markiert wird. Sei p die Wahrscheinlichkeit dieses Ereignisses. Dann ist

$$\begin{aligned} p &\leq \sum_{u \in \text{ Gefährlich}(v)} \text{prob}[\ u \text{ wird markiert }] = \sum_{u \in \text{ Gefährlich}(v)} \frac{1}{2 \cdot d(u)} \\ &\leq \sum_{u \in \text{ Gefährlich}(v)} \frac{1}{2 \cdot d(v)} \leq \frac{d(v)}{2 \cdot d(v)} = \frac{1}{2} \end{aligned}$$

und das war zu zeigen. □

Der Begriff eines guten Knotens ist für die Analyse zentral.

Definition 2.33 Der Knoten v heißt gut, wenn mindestens $d(v)/3$ Nachbarn einen Grad von höchstens $d(v)$ besitzen. Eine Kante heißt gut, wenn mindestens ein Endpunkt gut ist.

Der Knoten v ist also gut, wenn genügend viele Nachbarn keinen größeren Grad als v besitzen. Dann sollte die Wahrscheinlichkeit groß sein, dass einer dieser recht vielen niedriggradigen Nachbarn w markiert wird: Ein Knoten ist umso wahrscheinlicher, je kleiner sein Grad ist. w wird dann mit Wahrscheinlichkeit $\frac{1}{2}$ in die Menge I aufgenommen und verhindert damit die Aufnahme von v. Wir formalisieren diese Überlegung: Mit Wahrscheinlichkeit

$$p = (1 - \frac{1}{2 \cdot d(v)})^{d(v)/3} \leq e^{-1/6}$$

wird keiner der $d(v)/3$ niedriggradigen Nachbarn von v markiert. Im komplementären Fall wird aber mindestens einer dieser Nachbarn markiert und dieses Ereignis tritt mit Wahrscheinlichkeit mindestens $1 - e^{-1/6}$ ein. Wir zeichnen einen beliebigen markierten Nachbarn w aus und erhalten mit Lemma 2.32, dass w mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ „überlebt“ und in I aufgenommen wird. Damit haben wir das folgende Zwischenergebnis erhalten:

Lemma 2.34 *Sei v ein guter Knoten mit mindestens einem Nachbarn. Dann wird v mit Wahrscheinlichkeit mindestens $\frac{1-e^{-1/6}}{2}$ in einer Runde eliminiert.*

Also verlieren wir im Erwartungsfall einen konstanten Prozentsatz aller guten Knoten. Aber leider kann es nur wenige gute Knoten geben, wie das anfangs betrachtete Beispiel eines vollständig bipartiten Graphen mit wenigen hochgradigen Knoten zeigt. Aber wir haben Glück, denn wir werden gleich sehen, dass die meisten Kanten gut sind und damit werden viele Kanten in einer Iteration „eliminiert“, da ein guter Endpunkt eliminiert wird.

Lemma 2.35 *Mindestens die Hälfte aller Kanten ist gut.*

Beweis: Nach Definition besitzt ein schlechter Knoten weniger als $d(v)/3$ Nachbarn von niedrigerem Grad und damit gibt es mindestens doppelt so viele höhergradige Nachbarn wie niedergradige Nachbarn.

Wir richten Kanten vom niedergradigen zum höhergradigen Endpunkt. Wenn eine (jetzt gerichtete) Kante (u, v) schlecht ist, dann können wir ihr *injektiv* zwei weitere Kanten (v, u_1), (v, u_2) zuweisen, denn der schlechte Knoten v hat ja mindestens doppelt so viele höhergradige Nachbarn u_1, u_2 wie niedergradige Nachbarn u.

Wir haben also eine Injektion g von der Menge der schlechten Kanten in die Menge aller Kanten konstruiert und g weist jeder schlechten Kante zwei Kanten zu. Dies ist aber nur dann möglich, wenn es höchstens $\frac{|E|}{2}$ schlechte Kanten gibt. □

Sei F die Menge aller Kanten vor einer Iteration und sei F' die Menge der nach der Iteration verbleibenden Kanten. Mindestens die Hälfte aller Kanten in F besitzt einen guten Knoten als Endpunkt. Ein guter Knoten wird aber mit Wahrscheinlichkeit mindestens $c = \frac{1-e^{-1/6}}{2}$ eliminiert und damit ist $E[\,|F'|\,] \leq (1 - \frac{c}{2}) \cdot |F|$.

Aufgabe 34
Zeige, dass Algorithmus 2.31 nach erwartet logarithmisch vielen Iterationen hält.

Satz 2.36 *Algorithmus 2.31 findet für Graphen mit n Knoten nach erwartet $O(\log_2 n)$ Runden eine nicht vergrößerbare, unabhängige Menge.*

Aufgabe 35
Beschreibe einen randomisierten parallelen Algorithmus, der in erwartet polylogarithmischer Zeit ein nicht erweiterbares Matching für einen gegebenen Graphen $G = (V, E)$ konstruiert. Benutze dabei den Independent Set Algorithmus als Unterprogramm.

Aufgabe 36
Gegeben sei ein Universum $U = \{e_1, \ldots, e_n\}$ und ein Mengensystem $\mathcal{M} = \{E_1, \ldots, E_k\}$ mit $E_i \subseteq U$ für $i = 1, \ldots, k$. Es sei $g : U \to \{1, \ldots, 2n\}$ eine Gewichtsfunktion, die jedem Element in U ein zufälliges uniform aus $\{1, \ldots, 2n\}$ gewähltes Gewicht zuweist. (Das Gewicht einer Menge $E_i \in \mathcal{M}$ sei $\sum_{e \in E_i} g(e)$.) Zeige

$$\text{prob}[\text{ Es gibt nur eine Menge maximalen Gewichts in } \mathcal{M} \,] \geq \frac{1}{2}.$$

Beachte, dass diese Aussage für beliebiges k und damit für bis zu 2^n Mengen gilt!
Hinweis: Betrachte ein beliebiges, aber festes $e_j \in U$. Wir nehmen zunächst an, dass g für alle Elemente in $U \setminus \{e_j\}$ festgelegt sei. Betrachte nun

$$g_j(E_i) := \sum_{e \in E_i \setminus \{e_j\}} g(e).$$

Es sei $G_j = \max\{g_j(E) | E \in \mathcal{M}, e_j \in E\}$ und $G'_j = \max\{g_j(E) | E \in \mathcal{M}, e_j \notin E\}$. Zeige zunächst, dass $prob[g(e_j) = G'_j - G_j] \leq \frac{1}{2n}$ gilt. Folgere daraus, dass $prob$[Mehr als ein Menge hat maximales Gewicht] $\leq \frac{1}{2}$ gilt.

Aufgabe 37
Wir nehmen an, dass es einen Las Vegas Algorithmus A gibt, der zu einem gegebenen Graphen $G = (V, E)$ das größte Independent Set findet. Allerdings funktioniert A nur, wenn es genau ein Independent Set größter

Mächtigkeit gibt. Besitzt G zwei oder mehr optimale Independent Sets, dann kann der Ausgabe von A nicht getraut werden. A laufe in erwarteter Zeit $poly(|V|)$.

A funktioniert also nur, wenn das Versprechen abgegeben wird, dass G ein eindeutiges größtes Independent Set besitzt.

Zeige, dass es dann einen randomisierten Algorithmus gibt, der in erwarteter polynomieller Zeit für einen beliebigen Graphen $G' = (V', E')$ ein größtes Independent Set mit Wahrscheinlichkeit $\geq 1 - 2^{-|V'|}$ findet.

Fazit: Das Independent Set Problem wird nicht dadurch einfacher, dass das Versprechen der Eindeutigkeit gegeben wird.

2.6 Die probabilistische Methode

In der Kombinatorik möchte man zum Beispiel feststellen, ob es Objekte mit bestimmten Eigenschaften gibt. Die explizite Konstruktion solcher Objekte stellt sich in vielen Fällen als äußerst komplex heraus, der Nachweis der Existenz ist hingegen in einigen Fällen recht einfach:

> Zeige nicht nur, dass es Objekte mit den geforderten Eigenschaften gibt, sondern zeige sogar, dass die meisten Objekte die geforderten Eigenschaften besitzen!

Dieses Verfahren wird die *probabilistische Methode* genannt, da ein zufällig ausgewürfeltes Objekt hochwahrscheinlich die geforderten Eigenschaft besitzen wird. Wir stellen die probabilitische Methode exemplarisch in der Behandlung des folgenden Problems vor:

> Gibt es Graphen mit nur sehr kleinen Cliquen wie auch mit nur sehr kleinen unabhängigen Mengen?

Um diese Frage zu beantworten, betrachten wir Zufallsgraphen über der Knotenmenge $V = \{1, \ldots, n\}$: Wir werfen für jede potenzielle Kante $\{u, v\}$ eine faire Münze und setzen die Kante ein, wenn das Ergebnis Wappen ist.

Wir fixieren eine Knotenmenge $X \subseteq V$ der Größe k und bestimmen die Wahrscheinlichkeit p_X, dass X eine Clique oder eine unabhängige Menge ist. Es ist

$$p_X = 2 \cdot 2^{-\binom{k}{2}},$$

denn entweder ist X eine Clique und alle $\binom{k}{2}$ Kanten sind vorhanden oder X ist eine unabhängige Menge und keine der $\binom{k}{2}$ Kanten ist vorhanden. Wir sind vor Allem an der Wahrscheinlichkeit p_k interessiert, dass ein Zufallsgraph G eine Clique der Größe k oder eine unabhängige Menge der Größe k besitzt. Wir erhalten

$$\begin{aligned} p_k &= \text{prob}[\text{ Es gibt } X \text{ und } X \text{ ist eine Clique oder eine unabhängige Menge der Größe } k\,] \\ &\leq \binom{n}{k} \cdot 2 \cdot 2^{-\binom{k}{2}} < \frac{n^k}{k!} \cdot \frac{2 \cdot 2^{k/2}}{2^{k^2/2}}. \end{aligned}$$

Wir setzen $k = 2 \cdot \log_2 n$ und erhalten $n^k = 2^{k^2/2}$. Da andererseits $2 \cdot 2^{k/2} < k!$ für $k \geq 3$, folgt somit $p_k < 1$ für $k \geq 3$: Es gibt somit Graphen, die nur Cliquen oder unabhängige Mengen der Größe höchstens $2 \log_2 n - 1$ besitzen. Wir haben somit die Existenz eines Objekts durch Zählen nachgewiesen. Bis heute sind keine expliziten Graph-Konstruktionen bekannt, die vergleichbare Ergebnisse liefern!

Aufgabe 38
Wir betrachten Zufallsgraphen $G_{(n,p)}$ mit n Knoten, wobei die Kanten unabhängig voneinander mit Wahrscheinlichkeit p eingesetzt werden. X_k sei die Anzahl der Cliquen der Größe k in $G_{(n,p)}$. Zeige, dass $E[X_k] = \binom{n}{k} p^{\binom{k}{2}}$ gilt.

Im Folgenden betrachten wir einen Graphen $G_{(n,1/2)}$. Bestimme ein möglichst großes k als Funktion von n, so dass $E[X_k] \geq 1$ für genügend großes n. Bestimme eine möglichst kleines k als Funktion von n, so dass $E[X_k] \to 0$ für $n \to \infty$.

Aufgabe 39
$G = (V, E)$ sei ein ungerichteter Graph mit $V = \{1, \ldots, n\}$. Betrachte den folgenden Algorithmus A:

```
K:= {1}
for i := 2 to n do
   if Knoten i ist adjazent zu allen Knoten in K, then K := K ∪ {i}
Ausgabe: K
```

- Zeige, dass A stets eine maximale Clique ausgibt. (Eine Clique K heißt maximal, falls es keinen Knoten i gibt, so dass $K \cup \{i\}$ ebenfalls eine Clique ist.)
- Gib Datenstrukturen für G und K an und zeige, dass die Laufzeit des Algorithmus $O(n^2)$ ist.
- Betrachte das Verhalten von A auf einem Zufallsgraphen $G_{(n,1/2)}$: q_k bezeichne die Wahrscheinlichkeit, dass der Algorithmus mit einer Clique der Größe k terminiert. Zeige, dass $q_k \leq \binom{n}{k}(1-2^{-k})^{n-k}$. Begünde deine Annahmen über Unabhängigkeiten.
- Es sei $k_0 = (1-\epsilon)\log_2 n$ mit einer Konstanten $\epsilon > 0$. Zeige, dass es eine Konstante C gibt, so dass $q_{k_0} \leq e^{-Cn^\epsilon}$ für alle genügend großen n gilt. Zeige somit, dass

$$\text{prob}[A \text{ gibt eine Clique der Größe} \leq (1-\epsilon)\log_2 n \text{ aus}] \to 0 \quad \text{für } n \to \infty.$$

Fazit: Der Greedy Algorithmus berechnet für Zufallsgraphen gute Approximationen.

Kapitel 3

Random Walks

In diesem Kapitel beschreiben wir zahlreiche Anwendungen von Random Walks:

- Simulated Annealing führt einen Random Walk auf den Lösungen eines Optimierungsproblems durch, um eine optimale Lösung zu bestimmen.
- Ein Random Walk auf einem ungerichteten Graphen erlaubt eine Bestimmung der Zusammenhangskomponenten auf logarithmischem Platz.
- Ein „sorgfältig kontrollierter“ Random Walk bestimmt erfüllende Belegungen von 3-Sat Formeln schneller als bisher bekannte deterministische Algorithmen.
- Die Suchmaschine Google berechnet den Pagerank einer Webseite über einen Random Walk auf dem World-Wide-Web.
- In der Volumenbestimmung konvexer Mengen legt man ein Gitter über die konvexe Menge und exploriert das Gitter mit Hilfe eines Random Walks.

3.1 Simulated Annealing

Wir betrachten Optimierungsprobleme des Typs

$$\text{minimiere } f(x), \text{ so dass das Prädikat } L(x) \text{ wahr ist.}$$

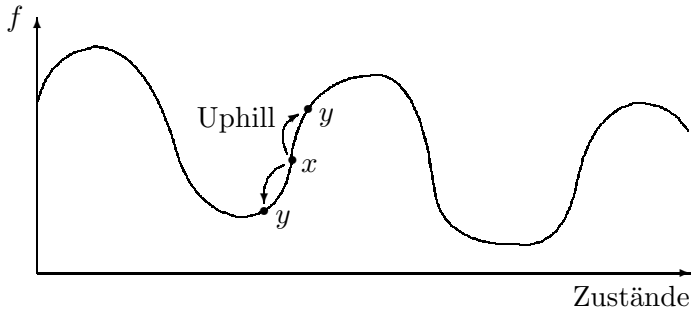
Dabei ist L ein Prädikat, dass genau dann wahr ist, falls x eine zulässige Lösung ist. Wir nehmen zusätzlich an, dass zu jeder Lösung x eine Umgebung $U(x)$ benachbarter Lösungen mit folgenden Eigenschaften existiert:

- Für jedes x gilt $x \in U(x)$ (d.h. x liegt in seiner eigenen Umgebung).
- Für alle x, y gilt: $x \in U(y) \Leftrightarrow y \in U(x)$.
- Für $x \in U(y)$ gilt $\mid U(x) \mid = \mid U(y) \mid$.

Eine solche Umgebung nennen wir symmetrisch. Simulated Annealing[1], also simuliertes Ausglühen, wählt zufällig einen Nachbarn y einer aktuellen Lösung x. Wenn $f(y) < f(x)$, dann

[1] Folgendes Beispiel erklärt den Begriff Simulated Annealing: Erhitzt man einen Stoff über seinen Schmelzpunkt hinaus und kühlt ihn dann behutsam ab, erhält man eine reine Kristallstruktur. Kühlt man zu schnell ab, treten kleine Fehler in der Kristallgitterstruktur auf. Diese fehlerhafte Struktur entspricht dann einem lokalen Energieminimum, während die reine Struktur ein globales Minimum darstellt.

wird x durch y ersetzt. Ist der Nachbar y nicht besser, also $f(y) \geq f(x)$, wird eine Ersetzung trotzdem mit sorgfältig berechneter Wahrscheinlichkeit akzeptiert. Mit dieser Stategie der *Uphill-Bewegung* versuchen wir, der „Falle“ eines lokalen Minimums zu entkommen.



Algorithmus 3.1 **Simulated Annealing**

(1) Die Zielfunktion $f(x)$ ist für Lösungen x zu minimieren.

(2) Setze $x = x_0$ für eine Anfangslösung x_0. Wähle eine Anfangstemperatur T.

Kommentar: Der Temperaturparameter T bestimmt unsere Bereitschaft, eine Uphill-Bewegung in Kauf zu nehmen.

(3) Wiederhole hinreichend oft:

(3a) Wähle zufällig einen Nachbarn $y \in U(x)$.

(3b) IF $f(y) \leq f(x)$ THEN $x = y$ ELSE setze $x = y$ mit Wahrscheinlichkeit $e^{-\frac{f(y)-f(x)}{T}}$.

(3c) Wähle eine neue Temperatur T.

Beachte, dass bei hoher Temperatur T die Ersetzung von x durch einen Nachbarn auch dann wahrscheinlich ist, wenn dieser Nachbar y einen großen Wert $f(y)$ hat, denn für $f(y)-f(x) << T$ ist:

$$e^{-\frac{f(y)-f(x)}{T}} \approx 1 - \frac{f(y)-f(x)}{T}.$$

Bei hoher Temperatur durchläuft Simulated Annealing den Lösungsraum durch jeweils zufällige Nachbarwahl. Bei geringer Temperatur wird eine Verschlechterung hingegen nur widerstrebend in Kauf genommen.

Wird die Temperatur nicht verändert, dann heißt das entsprechende Verfahren „**Metropolis Algorithmus**“. In Simulated Annealing wird der Metropolis Algorithmus also wiederholt mit sinkenden Temperaturen angewandt.

Wir betrachten das NP-vollständige Problem der Graph-Partitionierung, ein Problem, das daher wahrscheinlich keine effizienten Algorithmen besitzt.

Beispiel 3.1 **Graph-Partitionierung durch Simulated Annealing**
Im Problem der Graph-Partitionierung wird zu einem gegebenen, ungerichteten Graphen $G = (V, E)$ eine Knotenteilmenge $W \subseteq V$ mit $\mid W \mid = \frac{1}{2} \cdot \mid V \mid$ gesucht, so dass die Anzahl

$$\mid \{e \in E \ : \mid e \cap W \mid = 1\} \mid$$

kreuzender Kanten minimal ist. Eine Anwendung der Graph-Partitionierung ist der Entwurf von VLSI-Layouts, indem zuerst rekursiv Layouts für

$$(W, \{e \in E \mid e \subseteq W\}) \quad \text{und} \quad (V \setminus W, \{e \in E \mid e \subseteq V \setminus W\})$$

gesucht werden. Wenige kreuzende Kanten sind für die nachfolgende Kombination der beiden Layouts wichtig. Um Simulated Annealing auf die Graph-Partitionierung anzuwenden, lässt man beliebige Zerlegungen $(W, V \setminus W)$ zu und wählt

$$f(W) := \mid \{e \in E \ : \ \mid e \cap W \mid = 1\} \mid + \underbrace{\alpha \cdot (\mid W \mid - \mid V \setminus W \mid)^2}_{\text{Strafterm}}$$

als zu minimierende Funktion. Wir versuchen durch den Strafterm, eine perfekte Aufteilung, also $\mid W \mid = \frac{1}{2} \cdot \mid V \mid$, zu erzwingen.

Die Lösungsmenge ist die Potenzmenge von V. Als Startlösung für Simulated Annealing wählen wir ein perfekte, zufällig gewählte Aufteilung. Für eine Knotenteilmenge $W \subseteq V$ definieren wir die Umgebung von W als

$$U(W) := \{W' \subseteq V \mid \mid W \oplus W' \mid \leq 1\}.$$

$W' \in U(W)$ und W unterscheiden sich also höchstens um einen Knoten. Simulated Annealing versucht somit, eine gegebene Zerlegung $(W, V \setminus W)$ durch das Verschieben eines Knotens zu ändern.

In [JAGS] wird die Anfangstemperatur so gewählt, dass 40% aller Nachbarn akzeptiert werden. Die Temperatur wird über den Zeitraum von $16 \cdot \mid V \mid$ konstant gehalten und dann um den Faktor $0,95$ gesenkt („geometrische Abkühlung“). Bei zufällig gewählten Graphen schneidet Simulated Annealing (erstaunlicherweise?) erfolgreicher ab als maßgeschneiderte Algorithmen. Dieses Phänomen tritt auch auf, wenn wir den maßgeschneiderten Algorithmen die gleiche (große) Laufzeit wie der Simulated Annealing-Methode erlauben.

Weitere Experimente wurden für strukturierte Graphen durchgeführt. 500 (bzw. 1000) Punkte wurden zufällig aus dem Quadrat $[0,1]^2$ gewählt und zwei Punkte verbunden, wenn sie nahe beieinander lagen. Für diese Graphenklasse „bricht“ die Simulated Annealing-Methode ein und die maßgeschneiderten Algorithmen sind deutlich überlegen. Eine mögliche Erklärung für das unterschiedliche Abschneiden könnte in der unterschiedlichen Struktur der lokalen Minima liegen. Die geometrisch generierten Graphen werden Minima mit weitaus größeren Anziehungsbereichen als die zufällig generierten Graphen haben. Diese „Sogwirkung“ lokaler Minima ist bei den geometrisch generierten Graphen schwieriger zu überwinden.

Wir beginnen unsere Untersuchung des Verhaltens von Simulated Annealing und wählen zuerst die folgende Notation.

- $\mathcal{L}$ sei die Menge aller Lösungen.
- $G_T[i,j]$ sei die Wahrscheinlichkeit, dass Lösung j bei aktueller Lösung i und gegenwärtiger Temperatur T gewählt wird.
- $A_T[i,j]$ sei die Wahrscheinlichkeit, dass j als neue Lösung akzeptiert wird, wenn T die gegenwärtige Temperatur ist. Mit anderen Worten,

$$A_T[i,j] = \begin{cases} 1 & f(j) \leq f(i) \\ e^{-\frac{f(i)-f(j)}{T}} & \text{sonst.} \end{cases}$$

$P_T[i,j]$ sei die Wahrscheinlichkeit, dass Simulated Annealing für eine vorgegebene Temperatur T von Lösung i zu Lösung j wechselt. Dann gilt für $i \neq j$ offensichtlich

$$P_T[i,j] = G_T[i,j] \cdot A_T[i,j]. \tag{3.1}$$

Der Fall $i = j$ muss gesondert behandelt werden, da das Verwerfen einer benachbarten Lösung auf die ursprüngliche Lösung i führt. Es ist

$$P_T[i,i] = 1 - \sum_{j, j \in U(i), j \neq i} P_T[i,j]. \tag{3.2}$$

Offensichtlich ist

$$G_T[i,j] = \begin{cases} 0 & j \notin U(i) \\ \frac{1}{\mid U(i) \mid} & \text{sonst.} \end{cases}$$

Eine Modellierung von Simulated Annealing durch eine Markoff-Kette liegt jetzt nahe.

Definition 3.2 Eine Markoff-Kette (G, P) besteht aus einem gerichteten Graphen $G = (V, E)$ und einer Übergangsmatrix P, deren Zeilen und Spalten mit Knoten aus V indiziert sind. Für jeden Knoten $v \in V$ gilt

$$\sum_{w \in V} P[v,w] = 1.$$

Zusätzlich ist $P[v,w] = 0$ für $(v,w) \notin E$ und $P[v,w] \geq 0$ gilt stets. (Eine Matrix P mit $P \geq 0$ und $\sum_j P[i,j] = 1$ für alle i heißt *stochastische Matrix*.)

Gegeben sei eine Markoff-Kette (G, P) mit $G = (V, E)$ und eine Anfangsverteilung $q = (q_i | i \in V)$ auf der Menge der Knoten, d.h. es ist $q_i \geq 0$ und $\sum_{i \in V} q_i = 1$. Die Markoff-Kette definiert „Random Walks“ auf dem Graphen G wie folgt. Man beginnt den Weg in einem zufällig gemäß q gewählten Knoten i. Der i-te Zeilenvektors $q^i = (P[i,j] \mid j \in V)$ definiert eine Verteilung auf den Nachbarn von i und ein Nachbar j wird gemäß q^i zufällig gewählt. Dieser Prozess wird sodann „hinreichend“ oft wiederholt. Wir interpretieren $P[i,j]$ also als die Wahrscheinlichkeit, in einem Schritt vom Knoten i zum Knoten j zu wandern.

Eine zentrale Frage ist die (zumindest approximative) Bestimmung der Wahrscheinlichkeit, Knoten aus einer vorgegebenen Knotenmenge $W \subseteq V$ zu erreichen. Eine wesentliche Komponente dieser Frage ist natürlich auch die Bestimmung der Geschwindigkeit mit der Knoten in W hochwahrscheinlich erreicht werden können.

Simulated Annealing definiert für jede Temperatur T eine Markoff-Kette $(\mathcal{L}, P_T)$, deren Zustände oder Knoten den Lösungen entsprechen. Um das Verhalten von Simulated Annealing beurteilen zu können, müssen wir die nach k Schritten berechnete Lösung untersuchen. Da Simulated Annealing ein randomisiertes Verfahren ist, müssen wir die Wahrscheinlichkeiten

$$p_{i,j}^{(k)} = \left[\begin{array}{l} \text{Wahrscheinlichkeit, dass Lösung } j \text{ nach } k \\ \text{Schritten von Lösung } i \text{ aus erreicht wird} \end{array} \right]$$

untersuchen.

Lemma 3.3 *Es gilt* $p_{i,j}^{(k)} = P_T^k[i,j]$.

Beweis: Wir führen eine Induktion nach k. Die Verankerung für $k = 1$ ist offensichtlich. Nach Induktionsannahme gilt

$$\begin{aligned} p_{i,j}^{(k+1)} &= \sum_{r \in \mathcal{L}} \text{prob}\left[\begin{array}{l}\text{In } k \text{ Schritten wird Lösung} \\ r \text{ von } i \text{ aus erreicht}\end{array}\right] \cdot \text{prob}\left[\begin{array}{l}\text{In einem Schritt wird Lösung} \\ j \text{ von } r \text{ aus erreicht}\end{array}\right] \\ &= \sum_{r \in \mathcal{L}} P_T^k[i,r] \cdot P_T[r,j]. \end{aligned}$$

Wir erhalten

$$p_{i,j}^{(k+1)} = \sum_r P_T^k[i,r] \cdot P_T[r,j] = P_T^{k+1}[i,j]$$

und das war zu zeigen. □

Die Wahrscheinlichkeit, in einen zufälligen Weg bei Startverteilung q in k Schritten einen Knoten j zu erreichen, ist also durch die j-te Komponente von $q \cdot P_T^k$ gegeben. Wir sind an der Wahrscheinlichkeit $\lim_{k\to\infty} p_{i,j}^{(k)}$ interessiert. Wenn wir „Glück haben“, ist diese Grenzwahrscheinlichkeit unabhängig vom Startpunkt i.

Definition 3.4 Die Markoff-Kette (G, P) heißt genau dann ergodisch, wenn für alle i_1, i_2 und j die Grenzwerte $\lim_{k\to\infty} P^k[i_1, j]$ und $\lim_{k\to\infty} P^k[i_2, j]$ existieren und

$$\lim_{k\to\infty} P^k[i_1,j] = \lim_{k\to\infty} P^k[i_2,j] > 0$$

gilt. Die Verteilung $\pi^{(\infty)} = (\lim_{k\to\infty} P^k[1,j] | j \in V)$ heißt die Grenzverteilung von (G, P).

Für eine ergodische Markoff-Kette ist die Matrix

$$P^\infty = \lim_{k\to\infty} P^k = \left(\lim_{k\to\infty} P^k[i,j]\right)_{i,j\in V}$$

wohldefiniert, da wir voraussetzen, dass die Grenzwerte existieren. Weiterhin fordern wir, dass alle Zeilen in P^∞ identisch sind: Wenn wir einen Startknoten gemäß einer beliebigen Anfangsverteilung $\pi^{(0)}$ wählen und hinreichend viele zufällige Schritte machen, ist die Wahrscheinlichkeit einen bestimmten Knoten j zu erreichen, stets beliebig nahe an $\pi_j^{(\infty)} = P^\infty[1,j] = \cdots = P^\infty[n,j]$. Beachte, dass $\pi^{(\infty)}$ die Grenzverteilung ist.

Die Wahl des Anfangsknoten ist für einen hinreichend langen Random Walk also ohne Belang. Insbesondere gilt für jede Anfangsverteilung $\pi = (\pi_1, \ldots, \pi_N)$

$$\begin{aligned} \pi^T \cdot P^\infty &= \left(\sum_{i=1}^N \pi_i \cdot P^\infty[i,1],\ \ldots,\ \sum_{i=1}^N \pi_i \cdot P^\infty[i,N]\right) \\ &= \left(P^\infty[1,1] \cdot \sum_{i=1}^N \pi_i,\ \ldots,\ P^\infty[1,N] \cdot \sum_{i=1}^N \pi_i\right) \\ &= (\ P^\infty[1,j] \mid j = 1,\ldots,N) = \pi^{(\infty)} \end{aligned}$$

und es gilt also stets

$$\pi^T \cdot P^\infty = \pi^{(\infty)}. \tag{3.3}$$

Wir ersetzen π durch $\pi^{(\infty)}$ und erhalten

$$\begin{aligned}(\pi^{(\infty)})^T &= (\pi^{(\infty)})^T \cdot P^\infty = (\pi^{(\infty)})^T \cdot \lim_{k\to\infty} P^k \\ &= (\pi^{(\infty)})^T \cdot \lim_{k\to\infty} P^{k+1} = (\pi^{(\infty)})^T \cdot P^\infty \cdot P = (\pi^{(\infty)})^T \cdot P.\end{aligned}$$

$\pi^{(\infty)}$ ist aber auch eine Verteilung, denn es ist

$$\vec{1} = P^\infty \cdot \vec{1} = (\sum_j \pi_j^{(\infty)} \mid \cdots).$$

Definition 3.5 Sei (G, P) eine Markoff-Kette mit $G = (V, E)$. Dann heißt eine Verteilung $\pi = (\pi_v \mid v \in V)$ auf V eine stationäre Verteilung, falls $\pi^T \cdot P = \pi$.

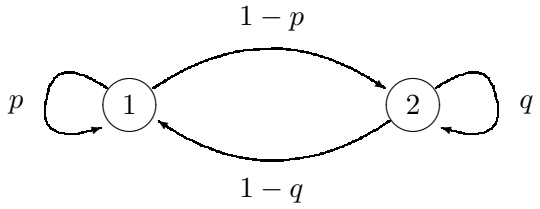
Wählen wir einen Startknoten gemäß einer stationären Verteilungen π und führen einen Schritt der Markoff-Kette durch, dann verbleiben wir in π.

Das folgende Lemma zeigt, dass die Grenzverteilung $\pi^{(\infty)}$ für ergodische Markoff-Ketten die einzige stationäre Verteilung ist. Wir können also die Grenzverteilung berechnen, indem wir das Gleichungssystem $\pi^T \cdot P = \pi$ lösen.

Lemma 3.6 *Jede ergodische Markoff-Kette (G, P) besitzt eine eindeutig bestimmte stationäre Verteilung π und es gilt $\pi = \pi^{(\infty)}$.*
Insbesondere stimmen stationäre Verteilung und Grenzverteilung also überein.

Beweis: Wir wissen bereits, dass $\pi^{(\infty)}$ eine stationäre Verteilung ist. Sei π eine beliebige stationäre Verteilung und wir erhalten $\pi = \pi^T \cdot P$ und damit natürlich auch $\pi = \pi^T \cdot P^\infty$. Andererseits ist aber $\pi^T \cdot P^\infty = \pi^{(\infty)}$ mit Gleichung (3.3) und wir haben die Eindeutigkeit einer stationären Verteilung nachgewiesen. □

Beispiel 3.2 Für welche Werte von p und q ist die folgende Markoff-Kette ergodisch?



Wir werden später einfache Kriterien angeben, um zu zeigen dass die Kette genau dann ergodisch ist, wenn $p, q < 1$ und $p + q > 0$ gilt. In diesen Fällen ist $1 - p, 1 - q > 0$, d.h. die Kanten von Knoten 1 zu Knoten 2 und zurück existieren, und es gibt mindestens eine Eigenschleife im Graphen.

Um die Grenzverteilung für $p, q < 1$ und $p + q > 0$ zu bestimmen, betrachten wir das Gleichungssystem

$$\begin{aligned}p\pi_1 + (1-q)\pi_2 &= \pi_1 \\ (1-p)\pi_1 + q\pi_2 &= \pi_2\end{aligned}$$

in den Unbekannten π_1 und π_2. Dieses System ist äquivalent zu $(1-p)\pi_1 = (1-q)\pi_2$. Da aber π eine Verteilung ist, also zusätzlich $\pi_1 + \pi_2 = 1$ zu fordern ist, erhalten wir

$$\begin{aligned} \pi_1 &= \frac{1-q}{1-p} \cdot \pi_2 \\ \pi_1 + \pi_2 &= \left(\frac{1-q}{1-p} + \frac{1-p}{1-p}\right) \cdot \pi_2 = \frac{2-p-q}{1-p} \cdot \pi_2 = 1. \end{aligned}$$

Die gesuchte stationäre Verteilung π ist also

$$\pi_1 = \frac{1-q}{2-p-q} \text{ und } \pi_2 = \frac{1-p}{2-p-q}.$$

Beachte, dass wir ausgenutzt haben, dass $1-p, 1-q > 0$ und dass damit $2-p-q > 0$ gilt. Falls $p = 1$ ist, gibt es keine Kante von Knoten 1 zu 2 und es ist $\lim_{k\to\infty} P^k[1,2] = 0$: Die Markoff-Kette ist nicht ergodisch. Auch im Fall $q = 1$ folgt, dass die Kette nicht ergodisch ist, da jetzt $\lim_{k\to\infty} P^k[2,1] = 0$ gilt.

Betrachten wir den Fall $p = q = 0$. Jetzt gibt es keine Eigenschleife im Graphen und man wandert in jedem Schritt von Knoten 1 zu 2 bzw. von 2 nach 1. Damit existieren die Grenzwerte nicht und folglich ist die Markoff-Kette in diesem Fall nicht ergodisch. Das Gleichungssystem besitzt jedoch die Lösung $\pi_1 = \pi_2 = \frac{1}{2}$, so dass wir eine nicht-ergodische Markoff-Kette mit stationärer Verteilung erhalten.

Aufgabe 40

Bei einer Irrfahrt auf den Zahlen $0, 1, \ldots, n+1$ wechselt ein Irrfahrer von einer Zahl i zur Zahl $i-1$ mit einer Wahrscheinlichkeit p_i und von i zur Zahl $i+1$ mit Wahrscheinlichkeit $q_i = 1 - p_i$. Wir sprechen von einer Irrfahrt mit reflektierenden Rändern, wenn $q_0 = 1 = p_{n+1}$.

Bestimme die stationäre Verteilung, falls $p_i = \frac{1}{2}$ für $1 \le i \le n$. **Bestimme** die stationäre Verteilung für den Fall, dass $p_i = p$ für $1 \le i \le n$ und ein $0 < p < 1$.

Bei der Irrfahrt mit reflektierenden Rändern 0 und $n+1$ sei jetzt zusätzlich für alle Zahlen $0 \le i \le n+1$ die Wahrscheinlichkeit r eingeführt, mit der der Irrfahrer bei der Zahl i bleibt. Mit Wahrscheinlichkeit $(1-r)q_i$ führt er eine zufällige Bewegung nach rechts, bzw. mit Wahrscheinlichkeit $(1-r)p_i$ nach links aus.

Bestimme nun die stationäre Verteilung. Für welche Werte von r ist die Markoff-Kette der Irrfahrt umkehrbar?

Aufgabe 41

Wir wissen, dass ergodische Markoff-Ketten eine eindeutig bestimmte stationäre Verteilung besitzen. Konstruiere eine Markoff-Kette, die mehr als eine stationäre Verteilung besitzt.

Zeige, dass jede Markoff-Kette, die zwei verschiedene stationäre Verteilungen besitzt, unendlich viele stationäre Verteilungen besitzt.

Aufgabe 42

Zeige, dass jede irreduzible Markoff-Kette höchstens eine stationäre Verteilung besitzt.

HINWEIS: Betrachte das Gleichungsystem $x^T \cdot P = x$ und $\sum_i x_i = 1$ für die Übergangsmatrix P der Markoff-Kette. Zeige, dass das System höchstens eine Lösung hat. Benutze dazu, dass der Eigenraum zum Eigenvektor 1 für die Matrix P^T dieselbe Dimension besitzt wie der Eigenraum zum Eigenvektor 1 der Matrix P.

Wir versuchen als nächstes, einfache Kriterien zu ermitteln mit deren Hilfe man die Ergodizität von Markoff-Ketten nachweisen kann. Zuerst beachten wir, dass in einer ergodischen Markoff-Kette (G, P) der Graph G stark zusammenhängend ist, d.h. zwischen je zwei Knoten existiert ein Weg.

Definition 3.7 Sei (G, P) eine Markoff-Kette mit $G = (V, E)$.

(a) (G, P) heißt irreduzibel, wenn es für alle Knoten i und j ein $k \in \mathbb{N}$ gibt, so dass $P^k[i,j] > 0$.

(b) Wir setzen für jeden Knoten $i \in V$

$$t_i = \text{ggt}(\{s \mid P^s[i,i] > 0\}).$$

Dann heißt der Knoten i aperiodisch, wenn $t_i = 1$ gilt. Für $t_i > 1$ heißt der Knoten i periodisch mit Periode t_i.

Die Markoff-Kette (G, P) heißt aperiodisch, wenn alle Knoten aperiodisch sind. Sie heißt periodisch, wenn alle Knoten periodisch sind.

Ergodische Markoff-Ketten sind irreduzibel, da die Grenzwerte $\lim_{k\to\infty} P^k[i,j]$ existieren und größer als 0 sind. Aus demselben Grund dürfen auch keine periodischen Knoten existieren: Ergodische Ketten sind irreduzibel und aperiodisch! Erstaunlicherweise gilt auch die Umkehrung.

Satz 3.8 *Eine Markoff-Kette ist genau dann ergodisch, wenn sie irreduzibel und aperiodisch ist.*

Beweis: Für den Beweis der Umkehrung verweisen wir auf Kapitel XV in Feller's Standardwerk [F]. □

Beachte, dass es Markoff-Ketten gibt, die weder periodisch noch aperiodisch sind, nämlich Markoff-Ketten, die Knoten beiden Typs besitzen. Eine jede Markoff-Kette zerfällt aber in irreduzible Markoff-Ketten, nämlichen die starken Zusammenhangskomponenten von G. Eine irreduzible Markoff-Kette ist aber entweder aperiodisch oder alle Knoten besitzen dieselbe Periode.

Wir geben eine äquivalente Betrachtungsweise des Begriffs der Periodizität an. Ein Knoten i ist periodisch (mit Periode t), wenn wir zum Zustand i nur nach einem Vielfachen von t Schritten zurückkehren können. Der Knoten i heißt aperiodisch, wenn keine solche Zahl $t > 1$ existiert. Insbesondere ist jeder Knoten mit $P[i,i] > 0$, also mit Eigenschleifen im zugehörigen Graphen, aperiodisch. Ist in diesem Fall die zugehörige Markoff-Kette auch irreduzibel, folgt bereits, dass die Kette aperiodisch ist. Für irreduzible periodische Markoff-Ketten mit Periode t kann man die Knoten V des zugehörigen Graphen disjunkt in Knotenmengen $V_0, \ldots, V_{t-1}$ zerlegen, so dass man bei einem Schritt von einem Knoten in V_i nur zu einem Knoten in $V_{i+1 \bmod t}$ übergehen kann.

Aufgabe 43
Sei (G, P) eine irreduzible Markoff-Kette.

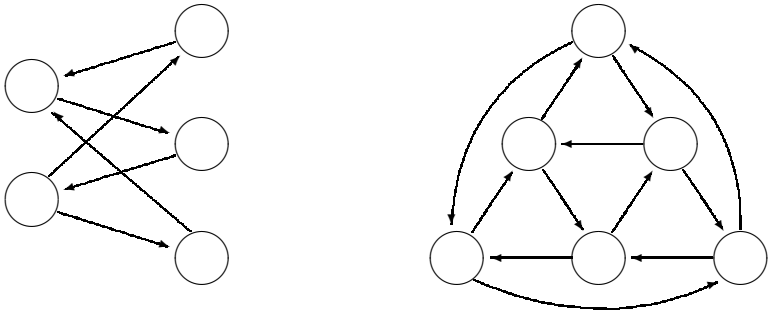
(a) Wenn die Knoten u bzw. v die Perioden p_u bzw. p_v besitzen, dann besitzt u die Periode $\text{ggt}(p_u, p_v)$.

(b) Wenn G einen aperiodischen Knoten besitzt, dann sind alle Knoten aperiodisch.

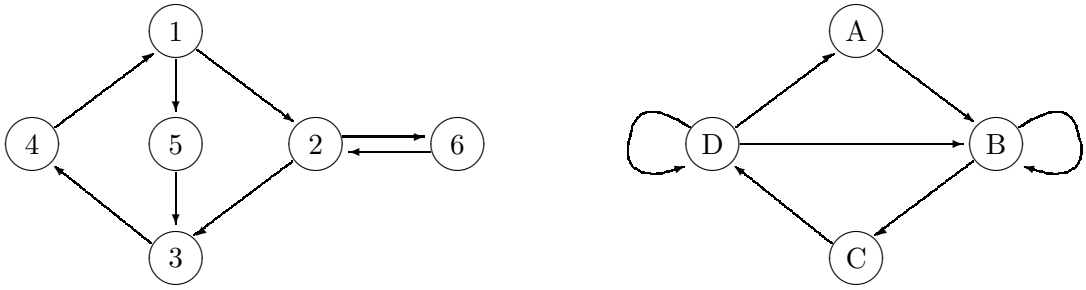
(c) Wenn G einen Knoten der Periode p besitzt, dann haben alle Knoten die Periode p.

Beispiel 3.3 In Beispiel 3.2 haben wir bereits gesehen, dass für $p + q > 0$ die Markoff-Kette aperiodisch ist, da der zugehörige Graph in mindestens einem Knoten eine Eigenschleife besitzt. Entfallen diese Eigenschleifen im Fall $p = q = 0$, dann ist der Graph bipartit und die Markoff-Kette besitzt die Periode 2. Beachte allgemein, dass jeder stark zusammenhängende bipartite Graph eine gerade Periode besitzt, da man die Knotenmenge V in zwei disjunkte Teilmengen V_0, V_1 zerlegen kann, so dass Kanten nur zwischen Knoten dieser beiden Mengen, aber nicht innerhalb der Mengen verlaufen.

Der bipartite Graph in der folgenden Abbildung besitzt beispielsweise die Periode 4, während der nicht-bipartite zweite Graph die Periode 3 hat.



Was sind die Perioden der beiden folgenden Graphen?



Ist die Markoff-Kette für Simulated-Annealing irreduzibel und aperiodisch? Wenn die Kette reduzibel ist, liegt ein Definitionsfehler der Nachbarschaft vor: Ein globales Minimum ist möglicherweise nicht von einer Anfangslösung erreichbar. Die Kette ist stets aperiodisch, denn jede Lösung ist auch ihre eigener Nachbar. Mit anderen Worten, jede vernüftige Implementierung von Simulated-Annealing führt auf eine ergodische Markoff-Kette und hat somit eine Grenzverteilung, die mit der eindeutig bestimmten stationären Verteilung zusammenfällt. Können wir diese Grenzverteilung bestimmen? Die folgende Vermutung drängt sich auf: Sei

$$q_T(x) = \frac{1}{N_T} \cdot e^{-\frac{f(x)}{T}}$$

mit der Normalisierung

$$N_T := \sum_{x \text{ Lösung}} e^{-\frac{f(x)}{T}}.$$

Dann ist q_T eine Verteilung, die guten Lösungen eine entsprechend hohe Wahrscheinlichkeit zuweist. „Entsprechend hoch“ wird dabei von der Temperatur gesteuert: Je niedriger die Temperatur, desto mehr dominiert der Beitrag $f(x)$ die Zielfunktion.

Um nachzuweisen, dass q_T tatsächlich die Grenzverteilung ist, benutzen wir den Begriff einer umkehrbaren Markoff-Kette.

Definition 3.9 Eine ergodische Markoff-Kette (G, P) heißt umkehrbar, wenn es eine Verteilung q gibt, so dass

$$q_i \cdot P[i,j] = q_j \cdot P[j,i] \tag{3.4}$$

für alle Knoten i und j gilt.

Das folgende Ergebnis zeigt, dass wir die Grenzverteilung über die Bedingung (3.4) bestimmen können, wenn eine umkehrbare Markoff-Kette vorliegt.

Korollar 3.10 *Sei (G, P) eine umkehrbare Markoff-Kette. Wenn eine Verteilung q für alle Knoten i und j die Bedingung (3.4) erfüllt, dann ist q die eindeutig bestimmte stationäre Verteilung von P und damit auch die Grenzverteilung der Markoff-Kette.*

Beweis: Wir fordern, dass die umkehrbare Markoff-Kette (G, P) ergodisch ist. Damit hat (G, P) nach Lemma 3.6 eine eindeutig bestimmte stationäre Verteilung, die mit der Grenzverteilung übereinstimmt. Es genügt der Nachweis, dass die Verteilung q mit Eigenschaft (3.4) stationär ist. Aus der Voraussetzung (3.4) folgt, dass für alle Knoten i gilt

$$\sum_j q_i \cdot P[i,j] = \sum_j q_j \cdot P[j,i].$$

Wegen

$$\sum_j q_i \cdot P[i,j] = q_i \sum_j P[i,j] = q_i$$

gilt

$$q_i = \sum_j q_j \cdot P[j,i]$$

und wir erhalten $q = q^T \cdot P$. □

Wir geben eine Interpretation der Umkehrbarkeitseigenschaft an. q ist die Grenzverteilung und q_i ist deshalb die Wahrscheinlichkeit, den Zustand i zu erreichen. Wie groß ist die Wahrscheinlichkeit, im vorigen Schritt in Knoten j gewesen zu sein (Ereignis vorher_j), gegeben, dass wir jetzt in Knoten i sind (Ereignis jetzt_i)? Es gilt

$$\text{prob}[\ \text{vorher}_j \mid \text{jetzt}_i\ ] = \frac{\text{prob}[\ \text{vorher}_j \wedge \text{jetzt}_i\ ]}{\text{prob}[\ \text{jetzt}_i\ ]} = \frac{q_j \cdot P[j,i]}{q_i}.$$

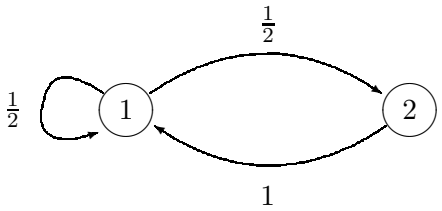
Unsere Forderung an umkehrbare Ketten ist also äquivalent zu

$$P[i,j] = \text{prob}[\ \text{vorher}_j \mid \text{jetzt}_i\ ]$$

und diese Forderung besagt, dass die Wahrscheinlichkeit $P[i,j]$ von i nach j zu wandern übereinstimmen muss mit der Wahrscheinlichkeit in j gewesen zu sein, wenn i erreicht wird: Die Kette ist tatsächlich „umkehrbar“.

Betrachten wir zuletzt symmetrische, ergodische Markoff-Ketten, also Ketten für die stets $P[i,j] = P[j,i]$ gilt. Offensichtlich(?) sind diese Ketten umkehrbar und wir erhalten aus Korollar 3.10, dass $q = (\frac{1}{n}, \ldots, \frac{1}{n})$ für $n = \mid V \mid$ die Grenzverteilung der Kette ist.

Beispiel 3.4 [Umkehrbare Markoff-Kette] Betrachte die Markoff-Kette aus Beispiel 3.2 für $p = \frac{1}{2}$ und $q = 0$.



Diese Kette ist ergodisch und besitzt die stationäre Verteilung $\pi = (\pi_1, \pi_2) = (\frac{2}{3}, \frac{1}{3})$ (warum?). Es ist

$$\pi_1 \cdot P[1,2] = \frac{2}{3} \cdot \frac{1}{2} = \frac{1}{3} = \pi_2 \cdot P[2,1]$$

Da für $i = j$ trivialerweise $\pi_i \cdot P[i,j] = \pi_j \cdot P[j,i]$ gilt, ist diese Markoff-Kette umkehrbar, obwohl sie nicht symmetrisch ist. Die Umkehrbarkeit besagt: Gegeben, dass wir im Knoten 2 sind, kamen wir durch den letzten Schritt vom Knoten 1 mit Wahrscheinlichkeit 1 bzw. vom Knoten 2 mit Wahrscheinlichkeit 0. Schließlich, gegeben, dass wir im Knoten 1 sind, kamen wir von Zustand 2 bzw. 1 jeweils mit Wahrscheinlichkeit $\frac{1}{2}$.

Wir können jetzt die von uns vermutete Grenzverteilung für das Simulated Annealing Verfahren verifizieren.

Satz 3.11 *Sei (G, P_T) die Markoff-Kette von Simulated-Annealing. Der Graph G sei stark zusammenhängend und die Umgebung sei symmetrisch. Dann ist*

$$q_T(x) = \frac{1}{N_T} \cdot e^{-\frac{f(x)}{T}}$$

die Grenzverteilung von (G, P_T).

Beweis: Nach Korollar 3.10 genügt der Nachweis von

$$q_T(i) \cdot P_T[i,j] = q_T(j) \cdot P_T[j,i].$$

In Gleichung (3.1) haben wir festgestellt, dass

$$P_T[i,j] = G_T[i,j] \cdot A_T[i,j].$$

Wir fordern eine symmetrische Nachbarschaft mit identischen Umgebungsgrößen und deshalb ist

$$G_T[i,j] = \frac{1}{\mid U(i) \mid} = \frac{1}{\mid U(j) \mid} = G_T[j,i].$$

Also genügt der Nachweis von

$$q_T(i) \cdot A_T[i,j] = q_T(j) \cdot A_T[j,i].$$

Die Behauptung folgt deshalb aus der folgenden Gleichungskette mit $[z]^+ := \max\{z, 0\}$

$$\begin{aligned} q_T(i) \cdot A_T[i,j] &= \frac{1}{N_T} \cdot e^{-\frac{f(i)}{T}} \cdot e^{-\frac{[f(j)-f(i)]^+}{T}} \\ &= \frac{1}{N_T} \cdot e^{-\frac{f(j)}{T}} \cdot e^{-\frac{f(i)-f(j)+[f(j)-f(i)]^+}{T}} \\ &= \frac{1}{N_T} \cdot e^{-\frac{f(j)}{T}} \cdot e^{-\frac{[f(i)-f(j)]^+}{T}} \\ &= q_T(j) \cdot A_T[j,i]. \end{aligned}$$

und das war zu zeigen. □

Was ist die Konsequenz von Satz 3.11? Zu opt := $\min\{f(x) \mid L(x)\}$ gilt offenbar

$$\lim_{T\to 0} q_T(i) = \lim_{T\to 0} \frac{e^{-\frac{-f(i)}{T}}}{\sum\limits_{j \text{ Lösung}} e^{-\frac{-f(j)}{T}}} = \lim_{T\to 0} \frac{e^{-\frac{f(i)-\text{opt}}{T}}}{\sum\limits_{j \text{ Lösung}} e^{-\frac{f(j)-\text{opt}}{T}}}.$$

Wenn $f(i) \neq$ opt, strebt der Zähler gegen Null. Der Nenner strebt hingegen gegen die Anzahl G der globalen Minima. Also gilt

$$\lim_{T\to 0} q_T(i) = \begin{cases} 0 & f(i) \neq \text{opt} \\ \frac{1}{G} & \text{sonst} f(i) = \text{opt} \end{cases}$$

und Simulated-Annealing wird bei **hinreichend langer** Laufzeit ein globales Minimum finden. Leider hat Simulated-Annealing auch bei „einfachen“ Optimierungsproblemen superpolynomielle Laufzeit, wie zum Beispiel beim Problem des maximalen Matchings (siehe [SH]). Zwar hat die Grenzverteilung von Simulated Annealing genau die von uns gewüschten Eigenschaften, leider ist aber die Konvergenzgeschwindigkeit in vielen Fällen viel zu gering: Die Zeit, die benötigt wird, um ein globales Optimum zu erreichen, ist viel zu hoch. Einen ersten Eindruck dieses Phänomens vermittelt das folgende Beispiel.

Beispiel 3.5 Wir nehem an, dass wir n Lösungen, nämlich die Elemente $1, 2, \ldots, n$ besitzen. Die Nachbarschaft von i sei $U(i) = \{i-1, i, i+1\}$, bzw. $U(1) = \{1, 2\}$ und $U(n) = \{n-1, n\}$. Wir setzen

$$f(x) = \begin{cases} 1 & x \leq n-1 \\ 0 & x = n \end{cases}$$

und nehmen an, dass wir in der Lösung 1 starten. Simulated Annealing wird jetzt einen Random Walk durchführen und hierbei von einem Knoten zu einem seiner drei beziehungsweise zwei Nachbarn gleichwahrscheinlich wandern. Nach einer erwarteten Anzahl von quadratisch vielen Schritten wird Simulated Annealing das Optimum $x = n$ erreichen, während eine vollständige Enumeration das Optimum bereits nach linear vielen Schritten findet.

Im allgemeinen ist nur das Erreichen suboptimaler Werte realistisch, verbunden mit der Hoffnung, dass diese eine gute Approximationen des Optimums darstellen. Simulated Annealing hat sich aber unter dieser abgeschwächten Erwartungshaltung bewährt und gehört zum Standardwerkzeug in der Optimierung.

Aufgabe 44

Theo Retiker geht jeden morgen zu Fuß ins Büro und jeden Abend zu Fuß nach Hause. Er besitzt drei Regenschirme, nimmt aber immer nur einen mit zur Arbeit oder mit nach Hause, wenn es gerade regnet. Befinden sich alle drei Regenschirme dort, wo er gerade nicht ist, und es regnet, so muss er durch den Regen gehen. Sei angenommen, dass es an jedem Morgen und jedem Abend mit der Wahrscheinlichkeit p regnet.

(a) Bestimme die erwartete Zahl von trocken zurückgelegten Wegen zwischen zwei mal Nasswerden in Abhängigkeit von p.

(b) Für welche Regenwahrscheinlichkeit p wird Theo am häufigsten nass?

Nehmen wir an, dass Theo gerne seltener nass werden würde. Er überlegt, ob er entweder weitere Schirme kaufen sollte oder gelegentlich aufräumen sollte.

(c1) Wie verändert sich bei fixierter Regenwahrscheinlichkeit die Zeit zwischen zweimal Nasswerden, wenn sich Theo k Regenschirme zulegt? Es kann angenommen werden, dass k gerade ist. Eine asymptotische Aussage genügt hier.

(c2) Wenn sich Theo einmal aufrafft und $k/2$ Regenschirme zu Hause und $k/2$ Regenschirme im Büro deponiert und dann wieder zu seinem gewohnten Verfahren zurückkehrt: Wie lange wird er nun in der Erwartung trocken unterwegs sein, bis er zum ersten mal wieder nass wird. Auch hier genügt eine asymptotische Aussage.

3.2 Speicherplatz-effiziente Suche in ungerichteten Graphen

Sei G ein ungerichteter Graph und G besitze die Knoten 1 und 2. Wir möchten überprüfen, ob die Knoten 1 und 2 zur selben Zusammenhangskomponente von G gehören.

Natürlich können wir die bekannten Traversierungsverfahren wie Tiefen- oder Breitensuche anwenden, aber unser Ziel ist ein speicherplatz-effizienteres Verfahren: Man beachte, dass der Stack der Tiefensuche oder die Schlange der Breitensuche im worst-case linear $\Theta(n)$ viele Knoten speichern muss, wobei n die Anzahl der Knoten des Graphen ist. Wir sind an einem Verfahren interessiert, das höchstens $O(\log_2 n)$ Speicherplatz benötigt.

Algorithmus 3.12 **Der Random-Walk Algorithmus**

(1) Die Eingabe besteht aus dem ungerichteten Graphen $G = (\{1, \ldots, n\}, E)$, wobei für jeden Knoten i die Eigenschleife $\{i, i\}$ eingesetzt sei.

T sei ein Schwellenwert.

(2) Setze $u = 1$ und $t = 0$.

(3) Solange $u \neq 2$ und $t \leq T$ wiederhole

(3a) Wähle zufällig eine Kante $\{u, v\}$.

(3b) Setze $u = v$ und $t = t + 1$.

(4) Wenn $u = 2$, dann akzeptiere und verwerfe ansonsten.

Wir benutzen also einen Random Walk, um den Eingabegraphen zufällig zu durchlaufen. Da der Graph ungerichtet ist, besteht nicht die Gefahr, dass wir uns in Sackgassen verirren, aber mit welcher Wahrscheinlichkeit werden wir einen Weg finden, wenn ein Weg existiert?

Die Markoff-Kette des Random Walk: Wir nehmen zuerst an, dass G zusammenhängend ist und ersetzen später G durch die Zusammenhangskomponente von Knoten 1. Die Zustände der Kette entsprechen den Knoten von G. Wir definieren die Übergangsmatrix P der Markoff-Kette durch

$$P[i,j] = \begin{cases} \frac{1}{d_i} & \{i,j\} \in E, \\ 0 & \text{sonst.} \end{cases}$$

d_i ist die Anzahl der Nachbarn von Knoten i, wobei i auch sein eigener Nachbar ist. Unsere Markoff-Kette ist irreduzibel, denn G ist zusammenhängend. Da jeder Knoten eine Eigenschleife besitzt, ist die Kette aperiodisch und damit ergodisch.

Wir behaupten, dass unsere Kette sogar umkehrbar ist. Dazu setzen wir $p_i = \frac{d_i}{2 \cdot |E|}$ und erhalten für jede Kante $\{i, j\}$

$$p_i \cdot P[i,j] = \frac{d_i}{2 \cdot |E|} \cdot \frac{1}{d_i} = \frac{1}{2 \cdot |E|} = \frac{d_j}{2 \cdot |E|} \cdot \frac{1}{d_j} = p_j \cdot P[j,i].$$

Die Kette ist also tatsächlich umkehrbar. Wie häufig wird eine Kante $e = \{i, j\}$ durchlaufen? Der Knoten i wird mit relativer Häufigkeit $p_i = \frac{d_i}{2 \cdot |E|}$ besucht. Die Wahrscheinlichkeit eines Besuchs von **e beginnend in i** ist deshalb

$$p_i \cdot \frac{1}{d_i} = \frac{d_i}{2 \cdot |E|} \cdot \frac{1}{d_i} = \frac{1}{2 \cdot |E|}.$$

Überraschenderweise wird somit jede Kante in jeder Richtung mit Wahrscheinlichkeit genau $\frac{1}{2\cdot|E|}$ durchlaufen!

Die Überdeckungszeit: Unser nächstes Ziel ist eine Abschätzung der Überdeckungszeit (engl. cover time), also der erwarteten Schrittzahl, die benötigt wird, um alle Knoten des zusammenhängenden Graphen G zu besuchen. Die Commute Time $C_{i,j}$ stellt sich hier als hilfreich heraus. $C_{i,j}$ ist die erwartete Schrittzahl, um von Knoten i nach Knoten j und zurück zu Knoten i zu wandern.

Wie groß ist die Commute Time, wenn $\{i,j\}$ eine Kante ist? Wir beachten, dass $C_{i,j}$ höchstens so groß ist wie die erwartete Zeit zwischen zwei Durchläufen der Kante $\{i,j\}$. Und als Konsequenz folgt

$$C_{i,j} \leq 2 \cdot |E|$$

Und wie groß ist die Überdeckungszeit? Da G zusammenhängend ist, besitzt G einen Spannbaum T (mit $n-1$ Kanten). Also gibt es einen Weg in G, der jede Kante von T zweimal, einmal in jeder Richtung, besucht. Wenn $e_1, \ldots, e_{n-1}$ die Kanten von T sind, dann ist die Überdeckungszeit also durch die Summe der Commute Times der Kanten, also durch $\sum_{i=1}^{n-1} C_{e_i} \leq 2 \cdot (n-1) \cdot |E|$ beschränkt.

Lemma 3.13

(a) Jede Kante von G wird in jeder Richtung mit Wahrscheinlichkeit genau $\frac{1}{2\cdot|E|}$ durchlaufen.

(b) Wenn $\{i,j\}$ eine Kante ist, dann ist $C_{i,j} \leq 2 \cdot |E|$.

(c) Die Überdeckungszeit für G ist höchstens $2 \cdot (n-1) \cdot |E|$.

Random Walk als speicher-effiziente Traversierung: Gibt es einen Weg von Knoten 1 nach Knoten 2? Wenn wir einen Random Walk in G, beginnend mit Knoten 1 durchführen und Knoten 2 antreffen, dann können wir mit Sicherheit sagen, dass Knoten 2 von 1 aus erreichbar ist. Haben wir hingegen nach $T = (2\cdot$ Überdeckungszeit) vielen Schritten Knoten 2 nicht gefunden, dann können wir mit der Antwort „Knoten 1 und 2 sind nicht verbunden“ abbrechen, handeln unser allerdings einen Fehler von höchstens 1/2 ein.

Satz 3.14 *Nach k unabhängigen Random Walks der Länge $T = 4n \cdot |E|$ wird eine existierende Verbindung von Knoten 1 nach Knoten 2 mit Wahrscheinlichkeit höchstens 2^{-k} nicht gefunden.*

Es werden also hochwahrscheinlich alle Knoten einer Zusammenhangskomponente besucht, solange der Random Walk die Länge $O(n \cdot |E|)$ besitzt. Der Random Walk Algorithmus ist also spürbar langsamer als Tiefensuche oder Breitensuche mit Laufzeit $O(n + |E|)$, erreicht aber logarithmischen Speicherplatz, während der Speicherplatz von Tiefen- und Breitensuche im worst-case linear ist.

Offenes Problem 1

Bisher ist es nicht bekannt, ob das Zusammenhangsproblem für ungerichtete Graphen auf logarithmischem Speicher durch deterministische Algorithmen lösbar ist.

Die nächste Aufgabe beschreibt universelle Traversierungssequenzen. Sollte eine universelle Traversierungssequenz auf logarithmischem Platz durch deterministische Algorithmen berechenbar sein, dann ist auch das Zusammenhangsproblem platz-sparend berechenbar.

Aufgabe 45
Eine universelle Traversierungssequenz für ungerichtete zusammenhängende Graphen mit n Knoten ist eine Folge von Elementen aus $\{1, \ldots, n\}$. Wenn man auf einem beliebigen zusammenhängenden Graphen von einem beliebigen Knoten aus startet, so soll bei Anwendung der Sequenz der ganze Graph durchlaufen werden. Dabei heißt Anwendung der Sequenz, dass in jedem Schritt in einem Knoten mit k Nachbarn das nächste Symbol i der Sequenz gelesen wird und zu Nachbar i mod k gegangen wird.

Zeige, dass universelle Traversierungssequenzen von polynomieller Länge existieren. Dabei muss eine solche Sequenz nicht effizient berechenbar sein.

Aufgabe 46
Gegeben ist ein zusammenhängender, ungerichteter Graph $G = (V, E)$ mit n Knoten und m Kanten. Eine Irrfahrt in G beginnt bei einem Knoten. Man wählt den Nachfolgeknoten jeweils gleichverteilt unter den Nachbarn des Knotens und setzt dann die Irrfahrt von dort fort. C_u sei die erwartete Zeit, um in einer Irrfahrt von u aus alle Knoten mindestens einmal zu besichtigen. $C(G) = \max_u C(u)$ heißt Überdeckungszeit.

G lässt sich als elektrisches Netzwerk betrachten, in dem jede Kante einem Widerstand von 1 Ohm entspricht. Es gelten Ohms Gesetz ($R = \frac{U}{I}$), sowie Kirchhoffs Regel (für jeden Knoten gilt, dass die einlaufenden Ströme gleich den auslaufenden sind). Der Widerstand zwischen zwei Knoten sei $R(u, v)$. Der Widerstand $R(G)$ eines Graphen G ist der maximale Widerstand zwischen zwei Knoten. Man kann $C_{u,v} = 2m \cdot R(u, v)$ beweisen. Zeige: $m \cdot R(G) \leq C(G)$.

Aufgabe 47
Betrachte folgende Graphen: P_n sei der Pfad aus n Knoten, L_n besteht aus dem vollständigen Graphen auf $\frac{1}{2}n$ Knoten sowie einem angefügten Pfad $P_{n/2}$, K_n sei der vollständige Graph auf n Knoten. Zeige:

(a) $C(P_n) = \Theta(n^2)$.

(b) $C(L_n) = \Theta(n^3)$.

(c) $C(K_n) = \Theta(n \log_2 n)$.

Zusätzliche Kanten helfen also nicht unbedingt.

Aufgabe 48
Am Ende eines anstrengenden Semesters beschließt Theo Retiker, dass er sich einen Urlaub verdient hat. Als erstes muss er in seiner Stadt die Autobahnauffahrt finden. Da er weiß, dass er bei zufälliger Irrfahrt durch die Stadt nach spätestens $O(|V| \cdot |E|)$ Schritten überall gewesen ist, möchte er sich die Mühe sparen, den Weg im Stadtplan zu suchen. Erst im letzten Moment fällt ihm ein, dass man zwar von jedem Punkt seiner Stadt aus jeden anderen erreichen kann, dass es aber Einbahnstraßen gibt.

Wir betrachten also Irrfahrten auf gerichteten, stark zusammenhängenden Graphen. Wenn die Irrfahrt einen Knoten erreicht, wird sie mit einer ausgehenden Kante, zufällig und gleichverteilt gewählt, fortgesetzt. Zeige, dass die erwartete Überdeckungszeit einer solchen Irrfahrt exponentiell in der Zahl von Knoten und Kanten wachsen kann.

3.3 Das k-Sat Problem

Im k-Sat Problem ist eine Formel α in konjunktiver Normalform gegeben, wobei die Klauseln von α aus höchstens k Literalen bestehen. Es ist zu entscheiden, ob α eine erfüllende Belegung besitzt.

Das k-Sat Problem ist NP-vollständig und effiziente Algorithmen sind deshalb nicht zu erwarten. Unser Ziel ist deshalb auch nur eine polynomielle Beschleunigung des enumerativen Verfahrens, dass bei n Variablen in Zeit $\text{poly}(n) \cdot 2^n$ alle Belegungen aufzählt und in die Formel einsetzt. Wir beginnen mit einem deterministischen Verfahren für 3-KNF. Wir nutzen aus, dass das 2-Sat Problem effizient lösbar ist.

Aufgabe 49
Entwickle einen effizienten Algorithmus, der entscheidet ob eine 2-Sat Formel erfüllbar ist.

Algorithmus 3.15

(1) Die Eingabeformel $\alpha \equiv k_1 \wedge \cdots \wedge k_N$ bestehe aus den Klauseln $k_1, \ldots, k_N$, wobei jede Klausel aus höchstens drei Literalen besteht. Setze $\mathcal{K} = \{k_1\}$.

(2) Solange es eine Klausel k_i gibt, die keine Variable mit einer Klausel in $\mathcal{K}$ gemeinsam hat, füge k_i zu $\mathcal{K}$ hinzu.

Kommentar: Je zwei Klauseln in $\mathcal{K}$ sind variablendisjunkt. Nach Berechnung von $\mathcal{K}$ hat jede Klausel k_i mindestens eine Variable mit einer Klausel in $\mathcal{K}$ gemeinsam.

(3) Bestimme die Menge $\mathcal{B}$ aller Belegungen, die alle Klauseln in $\mathcal{K}$ erfüllen.

Kommentar: Jede Klausel wird von sieben der acht möglichen Belegungen erfüllt. Wenn $\mathcal{K}$ aus m Klauseln besteht, dann ist $m \leq n/3$, denn je zwei Klauseln in $\mathcal{K}$ sind variablendisjunkt. Also besteht $\mathcal{B}$ somit aus höchstens $7^m \leq 7^{n/3}$ Belegungen. (Wir identifizieren zwei Belegungen, wenn sie sich nur in den Wahrheitswerten von Variablen unterscheiden, die nicht in einer Klausel aus $\mathcal{K}$ vorkommen.)

(4) Für alle Belegungen $b \in \mathcal{B}$: Setze b in α ein. Für jede Klausel ist mindestens eine vorkommende Variable bereits durch b gesetzt, und das resultierende 2-KNF Erfüllbarkeitsproblem ist zu lösen.

Satz 3.16 *Wenn $\mathcal{K}$ aus m Klauseln besteht, dann löst Algorithmus 3.15 das 3-Sat Problem in Zeit poly$(n+N) \cdot 7^m$ für Formeln mit n Variablen und N Klauseln.*
Die worst-case Laufzeit ist durch poly$(n+N) \cdot 7^{n/3} << 8^{n/3} = 2^n$ beschränkt.

Beweis: Warum finden wir eine erfüllende Belegung, wenn eine erfüllende Belegung b existiert? Wir durchlaufen alle Belegungen in $\mathcal{B}$, und wir werden deshalb auch b, eingeschränkt auf die in $\mathcal{K}$ auftauchenden Variablen, finden. Das entsprechende Erfüllbarkeitsproblem für 2-Sat ist jetzt aber lösbar, und wir werden tatsächliche eine erfüllende Belegung finden. □

Wir geben jetzt ein randomisiertes Verfahren an, das Belegungen zufällig auswürfelt und dann in wenigen Schritten versucht, die gewürfelte Belegung in eine erfüllende Belegung zu transformieren.

Algorithmus 3.17

(1) Die Eingabe $\alpha \equiv k_1 \wedge \cdots \wedge k_N$ sei eine k-KNF Formel mit n Variablen und N Klauseln.

(2) Wähle eine zufällige Belegung b.

(3) Wiederhole $3 \cdot n$-mal:

(3a) Akzeptiere α, wenn α durch b erfüllt wird.

(3b) Ansonsten bestimme eine Klausel k_i, die nicht durch b erfüllt wird.

(3c) Wähle zufällig eine in k_i vorkommende Variable und flippe ihren Wahrheitswert.

Bemerkung 3.1 Natürlich besteht die Gefahr, dass durch b erfüllte Klauseln durch Veränderungen in Schritt (3c) falsifiziert werden und dass der anscheinend zielgerichtete Random Walk zu einer zufälligen Irrfahrt entartet, wenn wir zu viele Schritte ausführen. Die entscheidende Beobachtung ist aber, dass eine erfüllende Belegung bereits durch den sehr kurzen Random Walk von Algorithmus 3.17 mit einer signifikant großen Wahrscheinlichkeit gefunden wird.

Im folgenden nehmen wir an, dass die k-KNF Formel α von der Belegung a erfüllt wird. $d(a,b)$ bezeichne den Hamming-Abstand zwischen einer Belegung b und der erfüllenden Belegung a.

Lemma 3.18 *α sei eine k-KNF Formel mit n Variablen. Dann findet Schritt (3) für $k \geq 3$ die Belegung a mit Wahrscheinlichkeit mindestens $\frac{1}{2} \cdot (\frac{1}{k-1})^{d(a,b)}$.*

Beweis: Sei b eine Belegung mit $d(a,b) = d$. Für die Analyse modellieren wir eine Iteration von Algorithmus 3.17 durch eine Markoff-Kette mit den Zuständen $0, 1, \ldots, d, \ldots, n$. Algorithmus 3.17 beginnt im Zustand d und wechselt von einem Zustand i zum Zustand $i-1$ (bzw. $i+1$) mit Wahrscheinlichkeit mindestens $\frac{1}{k}$ (bzw. mit Wahrscheinlichkeit höchstens $\frac{k-1}{k}$), da mindestens ein Literal der in Schritt (3b) ausgewählten Klausel auch von a erfüllt wird. Wir bestimmen die Wahrscheinlichkeit $q(b)$, dass ein Random Walk den Zustand 0 nach höchstens $3 \cdot d$ Schritten aufsucht.

Wenn ein Random Walk nach höchstens $3 \cdot d$ Schritten im Zustand 0 endet, wird er i falsche Entscheidungen getroffen haben, also einen mit a übereinstimmenden Wahrheitswert flippen. Ein solcher Random Walk wird insgesamt $d + 2 \cdot i$ Schritte lang sein, da die i falschen durch $d + i$ richtige Entscheidungen wettgemacht werden müssen. Also ist insbesondere $i \leq d$.

Sei Wege(i) die Anzahl der Random Walks, die zum ersten Mal nach genau $d + 2 \cdot i$ Schritten im Zustand 0 enden. Wir erhalten

$$q(b) \geq \sum_{i=0}^{d} \text{Wege}(i) \cdot \left(\frac{1}{k}\right)^{d+i} \cdot \left(\frac{k-1}{k}\right)^{i}.$$

Also genügt der Nachweis von

$$\sum_{i=0}^{d} \text{Wege}(i) \cdot \left(\frac{1}{k}\right)^{d+i} \cdot \left(\frac{k-1}{k}\right)^{i} \geq \frac{1}{2} \cdot \left(\frac{1}{k-1}\right)^{d}$$

oder äquivalent von

$$\sum_{i=0}^{d} \text{Wege}(i) \cdot \left(\frac{1}{k}\right)^{i} \cdot \left(\frac{k-1}{k}\right)^{d+i} \geq \frac{1}{2}. \tag{3.5}$$

Die linke Seite von (3.5) ist aber gerade die Wahrscheinlichkeit, dass ein Random Walk, der im Zustand d beginnt und mit Wahrscheinlichkeit $\frac{k-1}{k}$ nach links (bzw. mit Wahrscheinlichkeit $\frac{1}{k}$ nach rechts) wandert, den Zustand 0 nach höchstens $3 \cdot d$ Schritten erreicht. Die Behauptung folgt jetzt aus der nächsten Übungsaufgabe. □

Aufgabe 50
Ein im Zustand d beginnender Random Walk wird mit Wahrscheinlichkeit mindestens 1/2 im Zustand 0 nach höchstens $3 \cdot d$ Schritten enden, falls die Wahrscheinlichkeit nach links zu wandern mindestens 2/3 ist.

Wir können jetzt unser Hauptergebnis formulieren.

Satz 3.19 *α sei eine k-KNF Formel mit n Variablen und N Klauseln. Es gelte $k \geq 3$. Wenn α erfüllbar ist, dann wird eine erfüllende Belegung nach*

$$2 \cdot L \cdot \left(\frac{2 \cdot (k-1)}{k}\right)^{n} = 2 \cdot L \cdot \left(\frac{k-1}{k}\right)^{n} \cdot 2^{n}$$

Iterationen von Algorithmus 3.17 mit Wahrscheinlichkeit mindestens $1 - e^{-L}$ gefunden. Für 3-Sat wird somit eine erfüllende Belegung hochwahrscheinlich in Zeit poly$(n+N) \cdot \left(\frac{4}{3}\right)^{n}$ gefunden.

Beweis: Sei a eine beliebige erfüllende Belegung von α und $d(a,b)$ bezeichnen den Hamming-Abstand zwischen a und einer Belegung b. Wir bestimmen zuerst die Wahrscheinlichkeit p, dass eine Iteration von Algorithmus 3.17 erfolgreich ist. Mit Lemma 3.18 folgt

$$\begin{aligned} p &\geq \sum_{b\in\{0,1\}^n} \text{prob}[\ b \text{ wird gewählt }] \cdot \frac{1}{2} \cdot (\frac{1}{k-1})^{d(a,b)}. \\ &= \sum_{d=0}^{n} \text{prob}[\ \text{ eine Belegung } b \text{ mit } d(a,b)=d \text{ wird gewählt }] \cdot \frac{1}{2} \cdot (\frac{1}{k-1})^{d}. \end{aligned}$$

Wir beachten, dass es genau $\binom{n}{d}$ Belegungen b im Hamming-Abstand d von a gibt und erhalten deshalb mit dem binomischen Lehrsatz angewandt auf $(1+\frac{1}{k-1})^n$

$$\begin{aligned} p &\geq 2^{-n} \cdot \frac{1}{2} \cdot \sum_{d=0}^{n} \binom{n}{d} \cdot (\frac{1}{k-1})^{d} \\ &= 2^{-n} \cdot \frac{1}{2} \cdot (1+\frac{1}{k-1})^{n} \\ &= \frac{1}{2} \cdot \left(\frac{k}{2\cdot(k-1)}\right)^{n}. \end{aligned}$$

Wenn wir t Iterationen ausführen, dann ist die Misserfolgswahrscheinlichkeit höchstens

$$(1-p)^t \leq e^{-p\cdot t}$$

und die Misserfolgswahrscheinlichkeit für $t = \frac{1}{p}\cdot L$ ist durch e^{-L} beschränkt. □

Bemerkung 3.2 Wenn der Algorithmus die Formel α akzeptiert, dann besitzt α eine erfüllende Belegung. Wird hingegen keine erfüllende Belegung nach $2\cdot L\cdot\left(\frac{2\cdot(k-1)}{k}\right)^n$ Iterationen gefunden, dann ist α mit Wahrscheinlichkeit $1-e^{-L}$ nicht erfüllbar.

Aufgabe 51

Analysiere die erwartete Laufzeit des folgenden Algorithmus für das 2-SAT Problem unter der Annahme, dass eine erfüllende Belegung existiert.

Man beginnt mit einer beliebigen Belegung der Variablen. Dann wählt man eine unerfüllte Klausel, falls eine solche existiert und wählt sodann zufällig eine der beiden Variablen in der Klausel und ändert deren Wert. Dies wiederholt man so lange, bis eine erfüllende Belegung gefunden ist.

Wenn wir polynomielle Faktoren in der Laufzeit ignorieren, dann erreicht der deterministische Algorithmus die Laufzeit $O(7^{n/3})$. Da $7^{1/3} \approx 1,913$, haben wir mit der Laufzeit $(4/3)^n \approx (1,333)^n$ eine signifikante Beschleunigung erreicht. Eine leichte Verbesserung, nämlich von $(4/3)^n$ auf 1.3303^n wird in [HSSW] beschrieben. Diese Verbesserung wird erreicht, indem die zufällige Wahl einer Belegung b in Schritt (2) von Algorithmus 3.17 durch ein Preprocessing wie in Algorithmus 3.15 ersetzt wird.

3.4 Google

Suchmaschinen müssen für eine Anfrage, bestehend aus mehreren Stichworten, die informativsten Seiten für diese Stichworte suchen. Wir stellen im Folgenden den Random Walk Ansatz von Google vor.

Google bestimmt zuerst alle Webseiten, die die Stichworte der Anfrage enthalten. Die gefundenen Webseiten werden dann sowohl lokal wie auch global bewertet. In der lokalen oder anfrage-abhängigen Bewertung spielen Aspekte wie etwa

- Schriftgröße und Nähe der Stichworte zueienander innerhalb der Seite,

- Häufigkeit des Vorkommens der Stichworte innerhalb der Seite und

- Vorkommen der Stichworte in der Beschriftung von Hyperlinks, die auf die Seite zeigen

eine Rolle.

In der globalen oder anfrage-unabhängigen Bewertung wird der *Pagerank* der Webseite ermittelt. Der Pagerank $\mathrm{pr}(w)$ einer Webseite w soll die Qualität der Webseite wiedergeben und wird im Wesentlichen durch einen „Peer-Review“ bestimmt: Der Pagerank $\mathrm{pr}(w)$ ist hoch, wenn viele Seiten u mit hohem Pagerank $\mathrm{pr}(u)$ auf die Seite w zeigen.

Ein erster Ansatz zur Bestimmung des Pageranks nimmt an, dass eine Seite u mit d Hyperlinks $(u, v_1), \ldots, (u, v_d)$ jede Seite v_i mit $\frac{\mathrm{pr}(u)}{d}$ bewertet. Demgemäß erbt die Seite v_i den Beitrag $\frac{\mathrm{pr}(u)}{d}$ über den Hyperlink (u, v_i) und wir werden auf die Definition

$$\mathrm{pr}(w) = \sum_{u \text{ zeigt auf w}} \frac{\mathrm{pr}(u)}{d_u} \tag{3.6}$$

geführt, wobei wir annehmen, dass u genau d_u Hyperlinks besitzt. Um diese Definition zu interpretieren, fassen wir das WWW als eine Markoff-Kette auf: Die Knoten der Kette entsprechen den Webseiten und ein Hyperlink (u, w) entspricht einem Übergang von u nach w mit Wahrscheinlichkeit $1/d_u$. Wir erhalten somit die Übergangsmatrix P mit

$$P[u, w] = \begin{cases} 1/d_u & (u, w) \text{ ist ein Hyperlink,} \\ 0 & \text{sonst.} \end{cases}$$

Die Pagerank Definition (3.6) ist jetzt äquivalent zu

$$\mathrm{pr} = \mathrm{pr}^T \cdot P$$

und der Pagerank entspricht somit einer stationären Verteilung der WWW-Kette. Allerdings ist die WWW-Kette nicht irreduzibel, denn zum Beispiel sind Webseiten ohne Hyperlinks Sackgassen, und damit ist die Existenz einer stationären Verteilung nicht gesichert.

Deshalb verändert Google die Sichtweise ein wenig, um zu einer ergodischen Kette zu gelangen. Im erfolgreichen zweiten Ansatz legt Google einen Random Walk zugrunde, der mit Wahrscheinlichkeit $(1 - \alpha)$ eine benachbarte Seite aufsucht und mit Wahrscheinlichkeit α zu einer zufälligen Webseite springt. Wenn wir annehmen, dass es genau n Webseiten gibt, dann werden wir auf die Übergangsmatrix

$$P[u, w] = \begin{cases} \frac{\alpha}{n} + \frac{1-\alpha}{d_u} & (u, w) \text{ ist ein Hyperlink,} \\ \frac{\alpha}{n} & \text{sonst.} \end{cases}$$

geführt. Wir fordern wiederum, dass der Pagerank der (diesmal eindeutigen) stationären Verteilung entspricht und erhalten deshalb wegen $\mathrm{pr} = \mathrm{pr} \cdot P$ und $\sum_u \mathrm{pr}(u) = 1$ die Setzung

$$\begin{aligned} \mathrm{pr}(w) &= \sum_u \frac{\alpha}{n} \cdot \mathrm{pr}(u) + (1 - \alpha) \cdot \sum_{u \text{ zeigt auf w}} \frac{\mathrm{pr}(u)}{d_u} \\ &= \frac{\alpha}{n} + (1 - \alpha) \cdot \sum_{u \text{ zeigt auf w}} \frac{\mathrm{pr}(u)}{d_u}. \end{aligned}$$

Die anfrage-unabhängige Definition des Pageranks ist eine Schwäche von Google. Allerdings ist eine abfrage-abhängige Definition kaum vorstellbar, da dies eine Berechnung zum Zeitpunkt der Anfrage erfordert. Gegenwärtig werden mehrere Tausend PC's benötigt, die mehrere Stunden zur Berechnung des Pageranks benötigen und selbst dieser Aufwand ist erstaunlich gering, da mehrere Milliarden Webseiten involviert sind: Man nutzt die hohe Konnektivität des Webgraphen aus, die dazu führt, dass ein Random Walk sehr schnell gegen die stationäre Verteilung konvergiert. Wenn π_0 zum Beispiel eine Gleichverteilung auf den Webseiten ist, führt man sukzessive die Matrix-Vektor Produkte $\pi_{i+1} = \pi_i P$ aus und π_k wird für relativ kleines k bereits eine sehr gute Approximation der stationäen Verteilung darstellen, denn $\pi_k = \pi_0 P^k$.

3.5 Volumenbestimmung konvexer Mengen

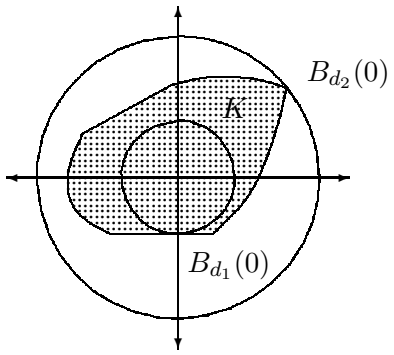
Sei K eine nichtleere, konvexe Menge im $\mathbb{R}^n$. Unser Ziel ist die approximative Bestimmung des Volumens von K. Wir „machen uns freiwillig das Leben schwer", indem wir annehmen, dass uns nur ein Orakel zur Verfügung steht, das Elementfragen „$x \in K$?" für rationale Vektoren $x \in \mathbb{Q}^n$ beantwortet. (Wir modellieren damit ein on-line Szenario, in dem ein Roboter in unbekanntem Terrain abgesetzt wird und der Roboter nur seine nächste Umgebung wahrnehmen kann.) Sei

$$B_r(c) = \{x \in \mathbb{R}^n \mid |x - c| \leq r\} \subseteq \mathbb{R}^n$$

eine Kugel mit Radius r und Mittelpunkt c. Wir setzen voraus, dass wir zur konvexen Menge K Parameter d_1 und d_2 mit

$$B_{d_1}(0) \subseteq K \subseteq B_{d_2}(0)$$

kennen. Selbst wenn diese Hilfestellung gegeben ist, haben I. Bárány und Z. Füredi [BF] gezeigt, dass eine approximative Volumenbestimmung für deterministische Algorithmen äußerst schwierig ist.



Fakt 3.1 *A sei ein deterministischer Algorithmus, der für jede konvexe Menge $K \subseteq \mathbb{R}^n$ mit*

$$B_{d_1}(0) \subseteq K \subseteq B_{d_2}(0).$$

untere und obere Schranken $\mathrm{vol}_{\mathrm{unten}}(K), \mathrm{vol}_{\mathrm{oben}}(K)$ *mit*

$$\mathrm{vol}_{\mathrm{unten}}(K) \leq \mathrm{volumen}(K) \leq \mathrm{vol}_{\mathrm{oben}}(K)$$

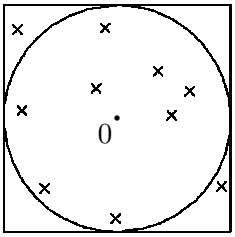
in Zeit $\text{poly}(n, \frac{1}{d_1}, d_2)$ *bestimmt. Dann gibt eine solche konvexe Menge* K_0 *mit*

$$\frac{\text{vol}_{\text{oben}}(K_0)}{\text{vol}_{\text{unten}}(K_0)} \geq \left(c \cdot \frac{n}{\log_2 n}\right)^n$$

für eine Konstante $c > 0$.

Für eine approximative Volumenbestimmung müssen wir also auf randomisierte Algorithmen zurückgreifen und hoffen, dass wir Algorithmen mit Laufzeit $\text{poly}(n, \frac{1}{d_1}, d_2)$ finden.

Beispiel 3.6 Wie kann man die Fläche des Einheitskreises approximativ bestimmen?



Wir ziehen zufällig eine genügend große Anzahl von Punkten $x_1, \ldots, x_L$ aus dem Quadrat $[-1, +1]^2$. Wenn L^* die Anzahl der Punkte aus dem Einheitskreis ist, dann wird

$$\frac{L^*}{L} \approx \frac{\text{volumen}(B_1(0))}{\text{volumen}([-1, +1]^2)}$$

gelten und $4 \cdot \frac{L^*}{L}$ wird eine gute Approximation der Kreisfläche π sein, da das Quadrat die Fläche vier hat.

Dieser Ansatz versagt aber im hoch-dimensionalen Fall: Das Volumen des n-dimensionalen Würfels $[-1, +1]^n$ ist 2^n, während das Volumen der Kugel mit Radius 1 durch

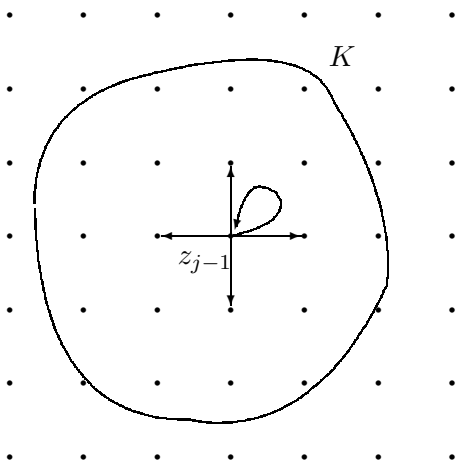
$$\text{volumen}(B_1(0)) = \frac{\pi^{\frac{n}{2}}}{\Gamma\left(1 + \frac{1}{2}n\right)} = \frac{2 \cdot \pi^{\frac{n}{2}}}{n \cdot \Gamma\left(\frac{1}{2}n\right)}$$

gegeben ist, wobei Γ die Gamma-Funktion mit $\Gamma\left(\frac{1}{2}\right) = \sqrt{\pi}$, $\Gamma(1) = 1$ und $\Gamma(x+1) = x \cdot \Gamma(x)$ für $x \in \mathbb{R}^+$ ist. Für eine geeignete Konstante $c > 1$ gilt

$$\text{volumen}(B_1(0)) \leq \frac{c^n}{\sqrt{n!}}.$$

Wenn wir somit nur polynomiell viele Vektoren zufällig aus dem Einheitswürfel $[-1, +1]^n$ ziehen, befindet sich darunter mit hoher Wahrscheinlichkeit kein Punkt der Kugel!

Wir werden eine Folge $r_0 = d_1, \ldots, r_N \geq d_2$ langsam wachsender Radien definieren, so dass sich die Volumina von $B_{r_{i-1}}(0) \cap K$ und $B_{r_i}(0) \cap K$ nur um höchstens den Faktor e unterscheiden. Sodann legen wir ein Gitter Γ über $K_i = B_{r_i}(0) \cap K$ und durchlaufen Γ $p(n)$-mal mit Random Walks, um zufällige Gitterpunkte aus K_i zu ziehen. Wenn e_i die Anzahl der Gitterpunkte ist, die bereits in K_{i-1} liegen, dann ist $v_i^* = \frac{p(n)}{e_i}$ eine gute Approximation für den Quotienten $v_i = \frac{\text{volumen}(K_i)}{\text{volumen}(K_{i-1})}$ und wir erhalten $\text{volumen}(K_0) \cdot v_1^* \cdot \cdots \cdot v_N^*$ als eine gute Approximation des Volumens von K.



Algorithmus 3.20 **Random-Walk von** $B_{r_i}(0) \cap K$

(1) Wähle $\epsilon = \left(\text{poly}(n, \frac{1}{d_1})\right)^{-1}$ und betrachte das Gitter

$$\Gamma = \{\epsilon \cdot x \mid x \in \mathbb{Z}^n\}.$$

Wir beginnen den Random-Walk in $z_0 = 0 \in \Gamma$.

(2) FOR $j = 1$ TO $\text{poly}(\frac{1}{d_1}, d_2, \frac{1}{\epsilon})$ DO

(2a) Wähle zufällig einen mit z_{j-1} benachbarten Gitterpunkt z, wobei z auch Nachbar von sich selbst ist.

(2b) IF $z \in B_{r_i}(0) \cap K$ THEN $z_j = z$ ELSE $z_j = z_{j-1}$.

Warum liefert aber Algorithmus 3.20 einen zufälligen Gitterpunkt aus $K_i := B_{r_i}(0) \cap K$? Wir modellieren den Random-Walk durch eine Markoff-Kette mit $K_i \cap \Gamma$ als Knotenmenge. (Beachte, dass $K_i \cap \Gamma$ eine endliche Menge ist). Wenn $z \in K_i \cap \Gamma$ insgesamt d Gitterpunkte als Nachbarn hat (inklusive sich selbst), erhält jeder solcher Übergang die Wahrscheinlichkeit $\frac{1}{d}$.

Die Markoff-Kette ist offensichtlich irreduzibel und aperiodisch und damit ergodisch. Wie sieht die eindeutig bestimmte stationäre Verteilung aus? Die Verteilung sollte „beinahe“ die Gleichverteilung sein, denn die relativ wenigen Randpunkte haben geringe Wahrscheinlichkeit. Lovász und Simonovits [LS] zeigen in einem technisch sehr aufwändigen Beweis, dass die Markoff-Kette “schnell mischt”, dass der Random Walk die stationäre Verteilung also schnell approximiert. Der vollständige Algorithmus hat jetzt die folgende Form.

Algorithmus 3.21 **Approximation des Volumens einer konvexen Menge**

(1) Die Eingabe besteht aus den Parametern d_1 und d_2 mit $B_{d_1}(0) \subseteq K \subseteq B_{d_2}(0)$.

(2) Definiere eine Folge von langsam größer werdenden Kugeln beginnend mit $B_{d_1}(0)$ und endend mit $B_{d_2}(0)$. Wähle als Radius der i-ten Kugel den Wert r_i mit

$$r_i = d_1 \cdot \left(1 + \frac{1}{n}\right)^i.$$

Wegen volumen$(B_r(0)) = r^n \cdot$ volumen$(B_1(0))$ gilt

$$\frac{\text{volumen}(B_{r_{i+1}}(0))}{\text{volumen}(B_{r_i}(0))} = \left(\frac{r_{i+1}}{r_i}\right)^n = \left(1+\frac{1}{n}\right)^n \leq e.$$

(3) Sei N minimal mit $r_N \geq d_2$.

Kommentar: Es genügt die Wahl $N \approx n \ln \frac{d_2}{d_1}$, da dann $r_N = d_1 \cdot (1+\frac{1}{n})^N \approx d_1 \cdot e^{\frac{N}{n}} = d_1 \cdot e^{\ln \frac{d_2}{d_1}} = d_2$.

(4) FOR $i = 1$ TO N DO

(4a) „Durchlaufe“ die konvexen Mengen $B_{r_i}(0) \cap K$ zufällig $p(n)$-mal mit Algorithmus 3.20 für ein Polynom p.

(4b) Wenn genau e_i Endpunkte der durchlaufenen Wege in $K \cap B_{r_{i-1}}(0)$ liegen, setzen wir

$$v_i^* = \frac{p(n)}{e_i}$$

als unsere Schätzung für den Quotienten

$$v_i = \frac{\text{volumen}(B_{r_i}(0) \cap K)}{\text{volumen}(B_{r_{i-1}}(0) \cap K)}.$$

(5) Gib volumen$(B_{d_1}(0)) \cdot v_1^* \cdot v_2^* \ldots v_N^*$ als Schätzung für das Volumen von K aus.

Wir erhalten eine gute Abschätzung von volumen$(B_{r_N}(0) \cap K) =$ volumen(K) mit Algorithmus 3.21, wenn die Einzelschätzungen v_i^* für v_i scharf sind, denn

$$\text{volumen}(B_{r_N}(0) \cap K) = \text{volumen}(B_{r_0}(0) \cap K) \cdot v_1 \cdot v_2 \ldots v_N.$$

Satz 3.22 *Algorithmus 3.21 approximiert das Volumen einer konvexen Menge $K \subseteq \mathbb{R}^n$ mit $B_{d_1}(0) \subseteq K \subseteq B_{d_2}(0)$ bis auf einen Faktor c mit Wahrscheinlichkeit $1-\delta$ in Laufzeit*

$$\textit{poly}(n, \frac{1}{d_1}, d_2, \log_2\left(\frac{1}{\delta}\right), \frac{1}{1-c}).$$

Beweis: Siehe [LS]. □

Der asymptotisch schnellste, bekannte Algorithmus zur Volumenberechnung einer konvexen Menge wird in [KLS] beschrieben. Es dürfte in der Praxis vorteilhafter sein, die konvexe Menge in größeren Schritten zu durchlaufen. Bei dem gegenwärtigen Punkt z_j wähle zum Beispiel eine zufällige Koordinate k und einen zufälligen Gitterpunkt aus

$$\{z_j + m\delta \cdot e_k \mid m \in \mathbb{Z}\} \subseteq \Gamma,$$

wobei e_k der Einheitsvektor mit einer Eins in Position k sei.

Kapitel 4

Pseudo-Random Generatoren

Wir haben die ganze Zeit über randomisierte Algorithmen geredet, wie aber gelangen wir an Zufallsbits? Physikalische Effekte wie das thermische Rauschen eines Widerstands oder radioaktive Zerfallsvorgänge liefern gute Zufallsgeneratoren. Allerdings benötigen wir Meßgeräte, die auf Schnelligkeit getrimmt werden müssen und auch die Reproduzierbarkeit eines Zufallsexperiments ist nicht gewährleistet.

In der Praxis verwendet man deshalb Pseudo-Random Generatoren, die aus einer „zufälligen Saat“, wie etwa der Systemzeit, eine zufällig erscheinende Bitfolge bauen. Der Konstruktionsprozess erfolgt durch einen deterministischen Algorithmus.

Gibt es gute Pseudo-Random Generatoren und was bedeutet es überhaupt, dass ein Pseudo-Random Generator gut ist? Um wieviel schneller können randomisierte Algorithmen im Vergleich zu deterministischen Algorithmen sein? Diese und ähnliche Fragen werden wir uns im Folgenden stellen.

4.1 One-way Funktionen und Pseudo-Random Generatoren

In der Simulation von randomisierten Algorithmen durch deterministische Algorithmen versucht man, einen Random Generator durch einen Pseudo-Random Generator zu simulieren. Ein Pseudo-Random Generatoren „streckt“ einen wirklichen Zufallstring und verlängert damit den String. Der gestreckte Zufallstring sollte sich wie ein wirklicher Zufallstring verhalten und damit alle relevanten statistischen Tests bestehen. Insbesondere sollten sich Random und Pseudo-Random Generatoren in effizient durchführbaren Tests nicht unterscheiden.

Definition 4.1 **(a)** Ein statistischer Test ist ein randomisierter Algorithmus, der eine Eingabe akzeptiert oder verwirft. Ein effizienter statistischer Test ist ein randomisierter Algorithmus mit polynomieller worst-case Laufzeit.

(b) Sei p ein echt monoton wachsendes Polynom mit $p(n) > n$. Ein Generator $G : \{0,1\}^* \to \{0,1\}^*$ mit Streckung p produziert auf binären Eingaben der Länge n binäre Ausgaben der Länge $p(n)$. G heißt effizient, wenn es eine in Polynomialzeit rechnende *deterministische* Turingmaschine gibt, die den Generator implementiert.

(c) Sei G ein Generator mit Streckung p und sei T ein statistischer Test. Wir setzen

$$\begin{aligned} g_n &= \text{prob}[\, T \text{ akzeptiert } G(x) \mid |x| = n \,] \text{ und} \\ r_n &= \text{prob}[\, T \text{ akzeptiert } y \mid |y| = p(n) \,]. \end{aligned}$$

G besteht den Test T, wenn es zu jedem $k \in I\!N$ eine Schranke N_k gibt, so dass

$$\forall n \geq N_k \;\; |g_n - r_n| \leq n^{-k}.$$

Die obigen Wahrscheinlichkeiten werden durch die Gleichverteilung auf den Paaren (x, u) (bzw (y, u)) definiert, wobei $x \in \{0,1\}^n$ (bzw. $y \in \{0,1\}^{p(n)}$) und u die Folge der vom statistischen Test T auf Eingabe $G(x)$ (bzw. y) angeforderten Zufallsbits ist. (Wir nehmen der Einfachheit halber an, dass alle Berechnungen von T auf Eingaben derselben Länge dieselbe Anzahl von Zufallsbits anfordern.)

Um einen Test T zu bestehen, darf sich der Generator also für große Eingabelängen nur unerheblich von einem Random-Generator unterscheiden, wobei der Test (bzw. Algorithmus) T zur Unterscheidung herangezogen wird.

Definition 4.2 Ein (effizienter) Generator G heißt ein (effizienter) Pseudo-Random Generator, wenn G jeden effizienten statistischen Test besteht.

Aufgabe 52

Eine alternative Definition von Pseudo-Random Generatoren lautet wie folgt: Sei G ein Generator mit Streckung p und sei T ein statistischer Test. Wir setzen

$$g_n = \text{prob}[\, G(x) \in T \mid |G(x)| = p(n) \,] \;\text{ und }\; r_n = \text{prob}[\, y \in T \mid |y| = p(n) \,].$$

G besteht den Test T, wenn es eine Schranke N gibt, so dass

$$\forall n \geq N \;\; |g_n - r_n| \leq 1/2.$$

Ein (effizienter) Generator G heißt ein (effizienter) Pseudo-Random Generator wenn G jeden statistischen Polynomialzeit-Test besteht.

Zeige, dass ein Pseudo-Random Generator gemäß der neuen Definition auch ein Pseudo-Random Generator gemäß der alten Definition ist.

Warum haben wir das doch schwache Konzept der effizienten statistischen Tests statt unbeschränkter statistischer Tests benutzt? Zum Einen sind wir ja am Einsatz von Pseudo-Random Generatoren für effiziente Algorithmen interessiert. Zum Anderen gibt es keine Pseudo-Random Generatoren, die alle statistischen Tests bestehen! Warum? Sei G ein Generator, der Eingaben der Länge n auf Ausgaben der Länge $p(n) > n$ strecke. Wir definieren den Test T mit

$$T(x) = \begin{cases} 1 & G \text{ produziert } x \text{ als Ausgabe} \\ 0 & \text{sonst.} \end{cases} \tag{4.1}$$

Offensichtlich ist prob[$G(x) \in T \mid |x| = n$] $= 1$, während ein Random Generator höchstens die Wahrscheinlichkeit $\frac{1}{2}$ erzielt.

Beispiel 4.1 Der auf linearen Kongruenzen basierende Generator arbeitet mit einer „Saat“ s, einem Modulus m, einem Koeffizienten a und einem Offset b. Die auf der Saat s basierende Ausgabe erfolgt gemäß der Rekursion

$$x_0 = s, \;\; x_{k+1} \equiv (a \cdot x_k + b) \mod p.$$

Dieser Generator wird häufig benutzt, ist aber leider **kein** Pseudo-Random Generator [B].

Ein weiteres Beispiel sind auf algebraischen Zahlen basierende Generatoren. (Eine algebraische Zahl ist die Nullstelle eines Polynoms mit ganzzahligen Koeffizienten.) Für eine algebraische Zahl α gibt dann der Generator G_α die Binärdarstellung von α aus. Leider ist auch dieser Generator **kein** Pseudo-Random Generator [KLL84].

Beachte, dass wir erlauben, eine Eingabe der Länge n nur um ein Bit zu strecken (denn $p(n) = n+1$ ist ein echt monoton wachsendes Polynom mit $p(n) > n$). Natürlich erhoffen wir uns eine Streckung um ein beliebig vorgegebenes Polynom.

Satz 4.3 *Sei G ein (effizienter) Pseudo-Random Generator und sei q ein beliebiges Polynom, das echt monoton wachse. Dann gibt es einen (effizienten) Pseudo-Random Generator G', der n Bits auf mindestens $q(n)$ Bits streckt.*

Beweis: Wir nehmen an, dass G n Bits auf $n+1$ Bits streckt; sollte G um mehr als $n+1$ Bits strecken, dann ignorieren wir die zusätzlichen Bits. Wir definieren einen neuen Bit Generator G^* durch

$$G^*(x) = G^{q(n)-n}(x).$$

Offensichtlich erreicht G^* die gewünschte Streckung und G^* ist effizient, wenn G effizient ist. Angenommen, G^* ist kein Pseudo-Random Generator. Dann gibt es also einen effizienten statistischen Test T^*, den G^* nicht besteht. Wir definieren

$$g_n^* = \text{prob}[\, T^* \text{ akzeptiert } G^*(x) \mid |x| = n \,] \quad \text{und} \quad r_n = \text{prob}[\, T^* \text{ akzeptiert } y \mid |y| = q(n) \,]$$

und erhalten, da G^* durchfällt,

$$|g_n^* - r_n| > n^{-k}$$

für unendlich viele n. Wir definieren die Wahrscheinlichkeit

$$p_i = \text{prob}[\, T^* \text{ akzeptiert } G^{q(n)-(n+i)}(y) \mid |y| = n+i \,]$$

für $0 \leq i \leq q(n) - n$ und erhalten

$$p_0 = g_n^* \quad \text{und} \quad p_{q(n)-n} = r_n.$$

Also gibt es ein i mit

$$|p_i - p_{i+1}| > \frac{n^{-k}}{q(n)}.$$

Wir definieren jetzt einen Test T, den der vermeintliche Pseudo-Random Generator G nicht besteht: Ein Wort der Länge $n+i+1$ ist genau dann zu akzeptieren, wenn der Test T^* das Wort $G^{q(n)-(n+i+1)}(x)$ akzeptiert.

Wenn wir G auf einen Random-String y der Länge $n+i$ anwenden, dann wird $G(y)$ genau dann akzeptiert, wenn T^* das Wort $G^{q(n)-(n+i+1)}(G(y)) = G^{q(n)-(n+i)}(y)$ akzeptiert und dies geschieht mit Wahrscheinlichkeit p_i.

Wenn wir hingegen mit einem Random-String $x \in \{0,1\}^{n+i+1}$ beginnen, dann akzeptiert T^* das Wort $G^{q(n)-(n+i+1)}(x)$ mit Wahrscheinlichkeit p_{i+1} und wir haben G von einem (wirklichen) Random Generator unterscheiden können, denn es ist ja $|p_i - p_{i+1}| > \frac{n^{-k}}{q(n)}$. □

Natürlich ist die Existenz von effizienten Pseudo-Random Generatoren die wesentliche Frage.

Lemma 4.4 *Wenn $NP \subseteq BPP$, dann gibt es keine effizienten Pseudo-Random Generatoren. Gleiches gilt im Fall von $P = NP$.*

Beweis: Der in (4.1) beschriebene Test T gehört zur Klasse NP und nach Annahme auch zur Klasse BPP. Folglich kann der Test, wie gefordert, durch einen effizienten randomisierten Algorithmus implementiert werden. Schließlich beachte, dass aus $P = NP$ die Bedingung $NP \subseteq BPP$ folgt. □

Die Existenz von Pseudo-Random Generatoren ist also alles andere als selbstverständlich und man wird Pseudo-Random Generatoren nur unter weiteren Annahmen wie $P \neq NP$ erhalten. Die Annahme $P \neq NP$ ist möglicherweise sogar zu schwach, denn sie ist eine Annahme über das worst-case Verhalten von Berechnungen, während die Existenz von Pseudo-Random Generatoren eine Frage des erwarteten Verhaltens von Berechnungen ist.

Definition 4.5 Eine Funktion $f : \{0,1\}^* \to \{0,1\}^*$ heißt eine one-way Funktion, wenn f in polynomieller Zeit durch eine deterministische Turingmaschine berechenbar ist und wenn für jedes Polynom p, für jede probabilistische Turingmaschine M, die in Zeit $O(p)$ rechnet, und für alle hinreichend großen n gilt, dass

$$\text{prob}[M \text{ berechnet auf Eingabe } f(x) \text{ ein } z \text{ mit } f(z) = f(x) \mid x \in \{0,1\}^n] < \frac{1}{p(n)}.$$

Die obige Wahrscheinlichkeit wird durch die Gleichverteilung auf den Paaren (x, u) definiert, wobei $x \in \{0,1\}^n$ und u die Folge der von M auf Eingabe $f(x)$ angeforderten Zufallsbits ist. (Wir nehmen der Einfachheit halber an, dass alle Berechnungen von M auf Eingaben der Länge n dieselbe Anzahl von Zufallsbits anfordern.)

Nicht nur ist die Annahme der Existenz von one-way Funktionen ausreichend, sondern diese Annahme ist sogar äquivalent zur Existenz von Pseudo-Random Generatoren.

Satz 4.6 *Die folgenden beiden Aussagen sind äquivalent:*

(a) Es gibt einen effizienten Pseudo-Random Generator.

(b) Es gibt one-way Funktionen.

Beweis (a) ⇒ (b): Sei G ein Pseudo-Random Generator. Mit Satz 4.3 können wir annehmen, dass G n Eingabebits auf $2n$ Ausgabebits streckt. Wir zeigen, dass G bereits eine one-way Funktion ist.

Wenn G keine one-way Funktion ist, dann gibt es ein k und eine probabilistische Turingmaschine M, so dass M eine Eingabe $y = G(x)$ mit Wahrscheinlichkeit mindestens n^{-k} „invertiert“. Wir definieren einen Test durch den folgenden randomisierten Algorithmus T: Simuliere M auf Eingabe y und akzeptiere y genau dann, wenn M ein x' mit $G(x') = y$ bestimmt.

Wenn wir einen Random String $y \in \{0,1\}^{2n}$ wählen, dann ist y mit einer Wahrscheinlichkeit von höchstens $\frac{2^n}{2^{2n}} = \frac{1}{2^n}$ Ausgabe von G auf einer Eingabe der Länge n.

Wählen wir hingegen y als eine Ausgabe von G, dann wird y mit einer Wahrscheinlichkeit von mindestens n^{-k} akzeptiert. Damit unterscheidet der Test zwischen G und einem wirklichen Zufallsgenerator und G ist im Widerspruch zur Annahme kein Pseudo-Random Generator.

Die Rückrichtung **(b) ⇒ (a)** ist weitaus schwieriger und wir verweisen auf [HILL]. □

Eine Reihe von vermuteten one-way Funktionen sind aus der Zahlentheorie bekannt.

Faktorisierung: Als Eingabe sind zwei Primzahlen p und q gegeben. Die Ausgabe ist das Produkt $N = p \cdot q$. Im Umkehrproblem ist also die Zahl N zu faktorisieren.

Das Problem des diskreten Logarithmus: Als Eingabe ist eine Primzahl p, ein erzeugendes Element g modulo p und eine natürliche Zahl i gegeben. Die Potenz $g^i \bmod p$ ist zu berechnen. Im Umkehrproblem sind p, g und $g^i \bmod p$ gegeben. Der „Logarithmus“ i ist zu berechnen.

Das RSA-Problem: Als Eingabe sind die Zahlen N, e und x gegeben, wobei e und $\phi(N)$, die Anzahl der primen Restklassen modulo N, teilerfremd seien. Der Modulus N ist ein Produkt von zwei (nicht bekannten) Primzahlen. Als Ausgabe ist $y \equiv x^e \mod N$ zu berechnen. Im Umkehrproblem ist x zu bestimmen, wobei y, N und e gegeben sind.

Das Problem der diskreten Quadratwurzelberechnung: Als Eingabe sind natürliche Zahlen m und $x < m$ gegeben. Die Ausgabe ist $(m, x^2 \bmod m)$. Das Umkehrproblem ist also die Bestimmung der Wurzel modulo m. (Eine effiziente Lösung gelingt effizient mit probabilistischen Algorithmen, solange m eine Primzahl ist.)

Weitere Kandidaten sind aus der Kombinatorik bekannt.

Das Dekodierproblem: Sei $\alpha < 1$. Als Eingabe ist eine $\alpha \cdot n \times n$ Matrix C von Nullen und Einsen gegeben. (Wir fassen die Zeilen von C als eine Basis des Coderaums auf, wobei der Coderaum durch die Linearkombinationen der Zeilen über dem Körper $\mathbb{Z}_2$ aufgespannt wird.) Weiterhin ist eine zu verschlüsselnde Nachricht $x \in \{0,1\}^{\alpha \cdot n}$ sowie ein Fehlervektor $e \in \{0,1\}^n$ (mit nicht zu vielen Einsen) vorgegeben. Die Ausgabe ist die verrauschte Kodierung $y = x^T \cdot C \oplus e$. Im Umkehrproblem ist die ursprünliche Nachricht x für das verrauschte Codewort y zu bestimmen.

Das Summen Problem für Teilmengen: Als Eingabe sind natürliche Zahlen $x_1, \ldots, x_n$ sowie eine Teilmenge $I \subseteq \{1, \ldots, n\}$ gegeben und die Ausgabe $y = \sum_{i \in I} x_i$ ist zu berechnen. Im Umkehrproblem ist aus den Zahlen $x_1, \ldots, x_n$ und dem Summenergebnis y eine Teilmenge $J \subseteq \{1, \ldots, n\}$ zu bestimmen, so dass $y = \sum_{j \in J} x_j$.

Um ein schwieriges Umkehrproblem zu „gewährleisten“, sollten die Zahlen aus dem Intervall $[0, 2^n]$ gewählt werden. Allerdings scheint das Summenproblem zu den schwächsten Kandidaten zu zählen.

Wir kommen als nächstes zu effizienten Generatoren, die vermutlich Pseudo-Random Generatoren sind.

Der Blum-Micali Generator wählt eine Primzahl p und eine erzeugende Restklasse g modulo p. Sodann wird für eine Saat s_0 die Iteration $s_{i+1} = g^{s_i} \bmod p$ berechnet und die Bitfolge $(b_1, \ldots, b_m)$ ausgegeben. Hierbei ist

$$b_i = \begin{cases} 1 & \text{wenn } s_i < p/2 \\ 0 & \text{sonst.} \end{cases}$$

Der BM-Generator basiert also auf dem diskreten Logarithmus als one-way Funktion.

Der RSA Generator: Für eine Saat s_0 berechnen wir $s_{i+1} = s_i^e \mod N$. Die Ausgabe des Generators für die Saat s_0 ist dann die Bitfolge $(s_1 \bmod 2, \ldots, s_m \bmod 2)$. Die Länge m der Folge sollte polynomiell in der Anzahl der Bits von s_0 sein.

Der Blum-Blum-Shub Generator: Für eine Saat s_0 berechnen wir $s_{i+1} = s_i^2 \bmod N$, wobei $N = p \cdot q$ mit Primzahlen $p \equiv q \equiv 3 \mod 4$ gelte. Die Ausgabe des Generators für

die Saat s_0 ist dann die Bitfolge $(s_1 \bmod 2, \ldots, s_m \bmod 2)$. Die Länge m der Folge sollte polynomiell in der Anzahl der Bits von s_0 sein.

Der BBS-Generator beruht also auf der diskreten Quadratwurzelberechnung als one-way Funktion.

4.2 Derandomisierung

Wir werden jetzt mit Hilfe von Pseudo-Random Generatoren eine „relativ gute“ Simulation von randomisierten Algorithmen durch deterministische Algorithmen erreichen. Wir benötigen allerdings eine stärkere Version von Pseudo-Random Generatoren.

Definition 4.7 **(a)** Ein nicht-uniformer statistischer Test ist ein randomisierter Algorithmus, der neben der Eingabe auch einen Orakelstring erhält, wobei der Orakelstring nur von der Eingabelänge, nicht aber von der Eingabe selbst abhängt.

Ein nicht-uniformer statistischer Test ist effizient, wenn der implementierende randomisierte Algorithmus eine polynomielle worst-case Laufzeit besitzt und wenn der Orakelstring von polynomieller Länge ist.

(b) Ein Generator $G : \{0,1\}^* \to \{0,1\}^*$ heißt ein starker Pseudo-Random Generator, wenn G alle nicht-uniformen, effizienten statistischen Polynomialzeit-Tests besteht.

Satz 4.8 *Wenn es einen effizienten, starken Pseudo-Random Generator gibt, dann ist*

$$BPP \subseteq \bigcap_{\varepsilon>0} \mathrm{DTIME}(2^{n^\varepsilon}).$$

Beweis: Sei M eine probabilistische Turingmaschine, die eine Sprache $L \in BPP$ erkenne. Wir nehmen an, dass M polynomielle Laufzeit besitzt.

Sei $k \in I\!N$. Satz 4.3 gilt auch für starke Pseudo-Random Generatoren und wir können annehmen, dass es einen effizienten starken Pseudo-Random Generator G gibt, der n Zufallsbits zu n^k Pseudo-Random Bits aufbläht.

Wenn M die Laufzeit $O(n^r)$ besitzt, dann fordert M höchstens $O(n^r)$ Zufallsbits an. Wir können somit M auf Eingabe w wie folgt deterministisch simulieren. Zuerst werden nacheinander alle $2^{O(n^{r/k})}$ Worte der Länge $n^{r/k}$ produziert. Wir fassen jedes produzierte Wort v als eine Saat auf, die wir durch den starken und effizienten Pseudo-Random Generator auf ein Wort v^* der Länge $O(n^r)$ strecken. Wir benutzen v^* als unser Reservoir von „Random-Bits“ und führen mit Hilfe dieser Bits eine deterministische Simulation von M durch. Schließlich akzeptieren wir die Eingabe w genau dann, wenn mehr als die Hälfte aller Simulationen akzeptieren.

Offensichtlich benötigt unsere Simulation $\mathrm{poly}(n) \cdot 2^{O(n^{r/k})}$ Schritte. Wir fixieren die Eingabe w und erhalten einen effizienten Test mit Orakelstring w wie folgt: Simuliere M auf Eingabe y und w; akzeptiere genau dann, wenn M „seine“ Eingabe w mit y als Zufallssequenz akzeptiert.

Da G ein starker Pseudo-Random Generator ist, besteht G den Test und unsere deterministische Simulation gelingt! □

Dieses Ergebnis läßt sich verbessern. Sei E die Komplexitätsklasse aller Sprachen, die durch deterministische Turingmaschinen in Zeit $2^{O(n)}$ erkannt werden können.

Satz 4.9 *[IW97] Wenn es eine Funktion $f \in E$ gibt, so dass f nur durch Schaltkreise der Größe $2^{\Omega(n)}$ berechenbar ist, dann folgt*

$$P = BPP.$$

Jede Sprache $L \in E$ besitzt eine uniforme Schaltkreisfamilie der Größe $2^{O(n)}$ und jede uniforme Schaltkreisfamilie der Größe $2^{O(n))}$ berechnet eine Sprache in E. Damit gilt also $P = BPP$, es sei denn Nicht-Uniformität beschleunigt die Berechnung einer jeden Funktion in E.

Teil II

On-line Algorithmen

Kapitel 5

Der Wettbewerbsfaktor

On-line Algorithmen geben die i-te Ausgabe, nachdem sie die ersten i Eingaben gelesen haben. Im Gegensatz dazu können off-line-Algorithmen zunächst die gesamte Eingabe lesen und dann alle Ausgaben produzieren.

Definition 5.1 Sei A ein Algorithmus, der für eine Eingabe $\sigma = (\sigma_1, \ldots, \sigma_n)$ eine Ausgabe $\tau = (\tau_1, \ldots, \tau_n)$ berechnet.

(a) A ist ein off-line Algorithmus, falls die Ausgabe berechnet wird, nachdem die gesamte Eingabe betrachtet wurde.

(b) A ist ein on-line Algorithmus, falls für jedes i nur die Eingabe $(\sigma_1, \ldots, \sigma_i)$ während der Berechnung von τ_i bekannt ist. Wir bezeichnen die Ausgabe τ_i mit $A(\sigma_1, \ldots, \sigma_i)$.

Wie sollen wir die Qualität einer on-line Strategie bewerten? Ein Vergleich mit off-line Strategien scheint unfair, da sich dann Algorithmen, die die Zukunft nicht kennen, an all-wissenden Algorithmen messen lassen müssen. Überraschenderweise lassen sich aber für viele wichtige algorithmische Fragestellungen tatsächlich on-line Algorithmen entwickeln, die mit off-line Algorithmen mithalten können!

Definition 5.2 Ein on-line Problem bewertet für jede Eingabefolge $\sigma = (\sigma_1, \ldots, \sigma_n)$ Ausgabefolgen $\tau = (\tau_1, \ldots, \tau_n)$. Die Bewertung erfolgt durch eine Funktion f mit Wert

$$f(\sigma, \tau) \in \mathbb{R}.$$

Ein off-line Algorithmus OPT ist optimal für f, falls

$$f(\sigma, \mathrm{Opt}(\sigma)) \leq f(\sigma, A(\sigma))$$

für jeden off-line Algorithmus A und für jedes σ gilt. Ein on-line Algorithmus A hat den Wettbewerbsfaktor[1] höchstens α für ein on-line Problem f, wenn es eine Konstante c gibt, so dass

$$f\big(\sigma, A(\sigma)\big) \leq \alpha \cdot f(\sigma, \mathrm{Opt}(\sigma)) + c$$

für alle Eingabefolgen σ gilt.

[1] In der Literatur wird der Begriff "competitive ratio" verwandt.

Unserer obigen Definition liegt zugrunde, dass wir Minimierungsprobleme mit einem möglichst guten on-line Algorithmus lösen möchten. Betrachten wir hingegen Maximierungsprobleme f, dann sagen wir, dass A den Wettbewerbsfaktor α hat, wenn

$$f\big(\sigma, A(\sigma)\big) \geq \frac{1}{\alpha} \cdot f(\sigma, \mathrm{Opt}(\sigma)) + c$$

für eine Konstante c gilt.

Wir haben sowohl für Minimierungs- wie auch für Maximierungsprobleme einen worst-case Vergleich zwischen der vorliegenden on-line Strategie und einer optimalen off-line Strategie durchgeführt. Natürlich machen auch andere Evaluierungen Sinn wie etwa ein Vergleich der erwarteten Qualität der on-line Strategie (bzgl. einer Wahrscheinlichkeitsverteilung auf Anfragefolgen) und der erwarteten Qualität der off-line Strategie: Aber welche Verteilung gibt die in der Praxis auftauchenden Anfragefolgen am besten wieder? An dieser Stelle wollen wir nur festhalten, dass ein worst-case Vergleich durchaus nicht „gott-gegeben“ ist.

Aufgabe 53

Professor Kalk hat vergessen, wo sein Auto geparkt ist. Dabei hat der Parkplatz die Form des Buchstaben V und Professor Kalk steht an der Spitze des V. Er kann sich nicht erinnern, auf welcher Seiten und wie weit entfernt von der Spitze das Auto geparkt ist. Professor Kalk betritt grundsätzlich nicht den Rasen zwischen den Seiten des V. Er kann sein Auto nur identifizieren, wenn er seinen Schlüssel probiert.

Gib eine Strategie an, mit der Professor Kalk sein Auto findet, und bei der sein Weg nur einen konstanten Faktor länger ist als der kürzeste Weg zum Auto.

Aufgabe 54

Theo befindet sich im Stop&Go-Verkehr auf einer zweispurigen Autobahn. Nehmen wir an, er startet auf der rechten Spur (Spur 0), die linke nennen wir Spur 1. Auf den beiden Fahrspuren geht es zu verschiedenen Zeiten unterschiedlich gut voran und immer wieder stellt sich die Frage, ob Theo versuchen sollte, die Spur zu wechseln. Dabei kann er natürlich nicht wissen, wie sich das Tempo einer Spur in der Zukunft entwickeln wird.

Er modelliert die Situation wie folgt. Er beginnt im Zustand 0. Allgemein befindet er sich im Zustand i und ihm wird ein 2-Tupel nichtnegativer ganzer Zahlen (s_0^k, s_1^k) gegeben, das angibt, wie stark man auf beiden Spuren gebremst wird. Nun muss Theo entscheiden, ob er entweder auf seiner Spur i bleibt, was die Verzögerungskosten von s_i^k verursacht, oder ob er in den Zustand $1-i$ wechselt, was die Kosten $d + s_{1-i}^k$ verursacht; d ist eine Konstante, die die Verzögerung durch einen Spurwechsel misst.

Der Stau ist irgendwann zu Ende, das heißt ab einem n_0 gilt für alle $n \geq n_0$, dass $s_0^n = s_1^n = 0$. Bis dahin will Theo so wenig Kosten wie möglich angehäuft haben, er kennt aber auch im Vorhinein das n_0 nicht. $OPT_i(k)$ stehe für die minimalen (Offline-)Kosten unter allen möglichen Verhaltensweisen, die in Spur i enden, falls der Stau nach Tupel k beendet wäre. Sei $OPT(k) = \min\{OPT_0(k), OPT_1(k)\}$ die beste beider Lösungen.

(a) Die situationistische Strategie sei folgende Regel: *Bleib, falls $s_i^k \leq s_{1-i}^k + d$, sonst wechsel die Spur.* Zeige, dass die situationistische Strategie keinen beschränkten Wettbewerbsfaktor hat.

(b) Die retrospektive Strategie sei folgende Regel: *Bleib, falls $OPT_i(k) \leq OPT_{1-i}(k)$, sonst wechsel die Spur.* Zeige, dass die retrospektive Strategie keinen beschränkten Wettbewerbsfaktor hat.

(c) Zeige: Falls es einen Algorithmus gibt, der einen Wettbewerbsfaktor von α erreicht, wenn $s_0^k = 0 \vee s_1^k = 0$ für alle k gilt, dann gibt es auch einen Algorithmus mit Wettbewerbsfaktor α für das Problem ohne die zusätzliche Einschränkung. Die Aussage kann bei der nächsten Teilaufgabe verwendet werden.

(d) Gib einen deterministischen Algorithmus **an**, der den Wettbewerbsfaktor 3 erreicht. Deterministische Algorithmen mit größeren, aber beschränkten Wettbewerbsfaktoren ergeben Teilpunkte.

(e) Gib einen effizienten Algorithmus **an**, der $OPT_0(k), OPT_1(k)$ und $OPT(k)$ berechnet.

5.1 Das Ski Problem

Wir fahren in den Skiurlaub und sind vor die Entscheidung gestellt, möglicherweise jeden Tag Skier für 1 Euro pro Tag zu leihen, oder für K Euro Skier zu kaufen. Leider wissen wir

nicht, wie lange die Skisaison dauert. Wenn wir am ersten Tag Skier kaufen, kann die Saison bereits am nächsten Tag beendet sein. Wenn wir jeden Tag mieten, hätten wir andererseits bei einer Saison von mindestens $K+1$ Tagen besser am ersten Tag Skier gekauft. Eine optimale Entscheidung ist unmöglich, da wir das Wetter während des Rests des Urlaubs nicht kennen. Gibt es zumindest eine vom Standpunkt des Wettbewerbsfaktors beste Strategie?

Die Eingaben für das Skierproblem sind Worte $\sigma \in 1^*0$, wobei die Anzahl der Einsen die Länge der Saison darstellt. Die Ausgabe sei

$$\tau_i \in \{\text{Miete am } i\text{-ten Tag}, \text{Kaufe am } i\text{-ten Tag}, \text{bereits gekauft}\}.$$

Im Skierproblem ist $f(\sigma, \tau_1, \ldots, \tau_n)$ der durch die Entscheidungen $(\tau_1, \ldots, \tau_n)$ ausgegebene Geldbetrag. Für Eingabefolgen $\sigma = 1^k0$ wird ein optimaler off-line-Algorithmus stets Skier mieten, falls $k < K$. Für $k \geq K$ werden Skier am ersten Tag gekauft.

Eine beste Strategie im Sinne von Definition 5.2 mietet Skier für die ersten $K-1$ Tage und kauft am K-ten Tag. Warum? Wir bestimmen zuerst den Wettbewerbsfaktor. Offensichtlich ist die Strategie optimal, wenn die Saison $K-1$ Tage oder weniger dauert. Bei K Tagen haben wir $K-1+K = 2 \cdot K-1$ Euro ausgegeben, während die beste Strategie nur K Euro kostet. Diese Strategie hat somit den Wettbewerbsfaktor $\frac{2 \cdot K-1}{K} < 2$ und ist besser als jede Strategie, die später kauft. Wenn hingegen eine Strategie am k-ten Tag mit $k \leq K-1$ kauft, gilt für den Wettbewerbsfaktor:

$$\alpha = \frac{k-1+K}{k} = 1 + \frac{K-1}{k} \geq 2.$$

Satz 5.3 *Wenn wir Skier am Tag K kaufen, dann erreichen wir einen Wettbewerbsfaktor zwei. Jede andere on-line Strategie hat einen Wettbewerbsfaktor größer als zwei.*

Wir greifen das Ski Problem wieder auf, fragen uns aber diesmal, ob randomisíerte on-line Strategien „mehr bringen“. Bevor wir uns dieser Frage zuwenden, müssen wir aber klären, wie eine randomisierte on-line-Strategie auszuwerten ist, denn schließlich wird nicht eine einzige Ausgabe, sondern vielmehr eine Verteilung über verschiedene Ausgaben geliefert.

Definition 5.4 Die Funktion f beschreibe ein on-line Problem. Wir sagen, dass eine randomisierte on-line Strategie A den Wettbewerbsfaktor höchstens α für F besitzt, wenn es eine Konstante c gibt, so dass

$$E[f(\sigma, A(\sigma))] \leq \alpha \cdot f(\sigma, \text{Opt}(\sigma)) + c$$

für jede Eingabefolge σ gilt.

$E[f(\sigma, A(\sigma))]$ ist der Erwartungswert der Zufallsvariablen $f(\sigma, A(\sigma))$. Diese Zufallsvariable bewertet die von A auf Eingabefolge σ zufällig erzeugte Ausgabe durch die Funktion f. Beachte, dass die Wahrscheinlichkeit einer bestimmten Ausgabefolge durch den Algorithmus A festgelegt wird, und $E[f(\sigma, A(\sigma))]$ ist nichts Anderes als die erwartete Bewertung der Ausgabe von $A(\sigma)$.

Wir benutzen hier das „Oblivious Adversary“ Modell: Der Gegner, der bestrebt ist, ein für die on-line Strategie möglichst schlechtes Szenario zu bestimmen, kennt zwar die on-line Strategie, kann aber das schlechte Szenario nicht sukzessive unter zusätzlicher Kenntnis der bisherigen Ausgaben der on-line Strategie „verschlimmern“.

Aufgabe 55
Das Spiel "Verkaufe den Rembrandt" hat folgende Regeln: Einem Spieler wird eine unendliche Sequenz von Preisen p_i nacheinander gegeben. In jedem Schritt i muss sich der Spieler entscheiden, ob der Preis p_i angenommen wird, woraufhin das Spiel mit Gewinn p_i endet.

Beim Spiel "Verkaufe dein Gold" gelten folgende Regeln: Der Spieler besitzt eine Menge Gold von G Kilogramm. In jedem Schritt bekommt er ein Preisangebot p_i für ein Kilo Gold. Er kann sich nun entscheiden zum Preis p_i pro Kilo einen Teil seines Goldes zu verkaufen. Das Spiel endet, wenn alles Gold verkauft ist, mit dem entsprechenden Gewinn. Zeige:

(a) A_2 sei ein deterministischer Algorithmus für "Verkaufe dein Gold". Dann gibt es einen randomisierten Algorithmus A_1 für "Verkaufe den Rembrandt", so dass bei jeder Folge von Preisen $(p_i \mid i)$ der erwartete Gewinn von A_1 auf der Folge $(p_i \mid i)$ gleich dem Gewinn von A_2 auf derselben Sequenz ist.

(b) A_1 sei ein randomisierter Algorithmus für "Verkaufe den Rembrandt". Dann gibt es einen deterministischen Algorithmus A_2 für "Verkaufe dein Gold", so dass bei jeder Folge von Preisen $(p_i \mid i)$ der erwartete Gewinn von A_1 auf der Folge gleich dem Gewinn von A_2 auf derselben Folge ist.

FAZIT: Beide Spiele sind "gleich schwierig", denn man kann sogar zeigen, dass ein randomisierter on-line Algorithmus für "Verkaufe dein Gold" nicht besser sein kann als der beste deterministische on-line Algorithmus.

Zurück zum Ski Problem: Eine randomisierte Strategie würde zu Beginn des $i+1$ten Tages mit Wahrscheinlichkeit π_i Skier kaufen (und nicht an vorangegangenen Tagen) und mit Wahrscheinlichkeit $1-\pi_i$ Skier mieten. Das Ziel ist die Bestimmung einer Verteilung π, so dass der Quotient aus erwartet entstehenden Kosten und minimalen Kosten möglichst klein ist. Dabei hat man sich vorzustellen, dass ein Gegner, der die Verteilung kennt, auf möglichst gemeine Art und Weise die Skisaison genau dann beendet, wenn dieser Quotient am größten ist.

Betrachten wir zuerst die erwarteten Kosten für eine Verteilung π. Es wäre unklug, Skier an einem Tag $t > K$ zu kaufen, da ein Kauf am Tag K geringere Kosten verursacht. Wir können deshalb o.B.d.A. annehmen, dass $\pi_t = 0$ für $t \geq K$ gilt. Wenn die Ski Saison am Ende des Tages T endet, dann betragen die erwarteten Kosten

$$E(T) = \sum_{t=0}^{T-1} (t+K) \cdot \pi_t + T \cdot \sum_{t=T}^{K} \pi_t,$$

denn wenn Skier an einem Tag $t+1 \leq T$ mit Wahrscheinlichkeit π_t gekauft werden, dann belaufen sich die Gesamtkosten auf $t+K$ da die Mietkosten t betragen. Werden Skier hingegen während der Saison nicht gekauft (und das geschieht mit Wahrscheinlichkeit $\sum_{t=T}^{K} \pi_t$), dann sind nur die Gesamtkosten T, also die Mietkosten zu bezahlen.

Da die Saison aber am Ende des Tages $T \leq K$ endet, betragen die optimalen Kosten T und wir erhalten die Bedingung

$$E(T) = \sum_{t=0}^{T-1} (t+K) \cdot \pi_t + T \cdot \sum_{t=T}^{K} \pi_t = \alpha \cdot T,$$

wenn wir den Wettbewerbsfaktor α erreichen wollen. Um eine möglichst gute Verteilung π zu berechnen, führen wir eine heuristische Rechnung durch: Wir nehmen an, dass die Variable T das reelle Intervall $[1, K]$ durchläuft, dass π eine auf $[1, K]$ definierte Verteilung beschreibt und erhalten dann die Bedingung

$$F(T) := \int_{t=0}^{T} (t+K) \cdot \pi(t) \cdot dt + T \cdot \int_{t=T}^{K} \pi(t) \cdot dt \leq \alpha \cdot T.$$

Wir nehmen weiterhin heuristisch an, dass Gleichheit besteht. Wir differenzieren beide Seiten nach T und erhalten die Bedingung

$$F'(T) = (T+K)\cdot\pi(T) + \left[\int_{t=T}^{K} \pi(t)\cdot dt - T\cdot\pi(T)\right] = \alpha \tag{5.1}$$

Das verbleibende Integral möchten wir ebenfalls loswerden und differenzieren nochmals nach T, um

$$\begin{aligned} F''(T) &= \left[\pi(T) + (T+K)\cdot\pi'(T)\right] + \left[-\pi(T) - \pi(T) - T\cdot\pi'(T)\right] \\ &= K\cdot\pi'(T) - \pi(T) = 0 \end{aligned}$$

zu erhalten. Die Differentialgleichung $\pi(T) = K\cdot\pi'(T)$ hat aber genau die Lösungen

$$\pi(T) = c\cdot e^{T/K}$$

für eine Konstante c, die so zu wählen ist, dass wir eine Verteilung erhalten: Da $(c\cdot K\cdot e^{T/K})' = c\cdot e^{T/K} = \pi(T)$ führt die Bedingung $1 = \int_0^K \pi(t)\cdot dt$ auf die Forderung $1 = c\cdot K\cdot e - c\cdot K$, und damit ist $c = \frac{1}{K\cdot(e-1)}$. Wie groß ist der Wettbewerbsfaktor α? Wir setzen die Lösung $\pi(T) = \frac{1}{K\cdot(e-1)}\cdot e^{T/K}$ in Bedingung (5.1) ein und erhalten für $T = 0$

$$\alpha = K\cdot\pi(0) + \int_{t=0}^{K} \pi(t)dt = \frac{1}{e-1} + 1 = \frac{e}{e-1}.$$

Satz 5.5 *Wenn Skier zu Beginn des Tages T mit Wahrscheinlichkeit $\pi_T = \frac{1}{K\cdot(e-1)}\cdot e^{T/K}$ gekauft werden, dann wird der Wettbewerbsfaktor $\frac{e}{e-1}$ erreicht.*

Da $\frac{e}{e-1} \approx 1,58$ haben wir im Vergleich zur optimalen deterministischen on-line Strategie eine signifikante Verbesserung erreicht. Der Leser ist aufgefordert das heuristische Argument durch einen formalen Beweis zu ersetzen. Tatsächlich ist unser randomisierte Algorithmus sogar optimal, denn

---

Aufgabe 56
Zeige, dass jeder randomisierte on-line Algorithmus mindestens den Wettbewerbsfaktor $\frac{e}{e-1}$ besitzt.

5.2 Scheduling

Wir beginnen mit dem **Minimum Makespan** Problem. Es sind n Aufgaben $A_1, \ldots, A_n$ gegeben, wobei Aufgabe A_i die Laufzeit t_i besitzt. Die Aufgaben sind so auf m Maschinen auszuführen, dass der „Makespan“, also die für die Abarbeitung aller Aufgaben anfallende Bearbeitungszeit, minimal ist. Mit anderen Worten, wenn

$$I_j = \{i \mid A_i \text{ wird auf Maschine } j \text{ ausgeführt}\}$$

die Menge der auf Maschine j auszuführenden Aufgaben ist, dann ist

$$\max_{1\le j\le m}\left\{\sum\nolimits_{i\in I_j} t_i\right\}$$

der Makespan. Wir benutzen eine denkbar einfache Greedy Strategie: Führe die aktuelle Aufgabe auf der Maschine mit der bisher geringsten Last aus.

Lemma 5.6 *Der Greedy Algorithmus besitzt den Wettbewerbsfaktor* 2.

Beweis: Für eine gegebene Instanz sei Maschine i die am schwersten „beladene“ Maschine, die also als letzte Maschine noch rechnet. Falls i nur eine Aufgabe ausführt, ist der on-line Algorithmus offensichtlich optimal. Angenommen, i führt mindestens zwei Aufgaben aus. Sei A_j die letzte von i ausgeführte Aufgabe. Wenn T die Gesamtlaufzeit von i ist, dann waren zum Zeitpunkt $T - t_j$ alle anderen Maschinen beschäftigt. Folglich ist

$$\sum_{k=1}^{n} t_k \geq (m-1) \cdot (T - t_j) + T = mT - (m-1) \cdot t_j \geq mT - m \cdot t_j$$

und damit folgt

$$T \leq \frac{1}{m} \cdot \sum_{k=1}^{n} t_k \quad + t_j \leq 2 \cdot \max\left\{ \frac{1}{m} \cdot \sum_{k=1}^{n} t_k,\ t_j \right\}.$$

Die Behauptung folgt mit der nächsten Aufgabe. □

Aufgabe 57
Für jede Aufgabe A_j ist der Makespan mindestens $\max\left\{ \frac{1}{m} \cdot \sum_{k=1}^{n} t_k,\ t_j \right\}$.

Unsere Analyse kann nicht verbessert werden wie das folgende Beispiel zeigt. $m \cdot (m-1)$ Aufgaben der Länge 1 sowie eine Aufgabe der Länge m sind gegeben. Offensichtlich lässt sich der Makespan m erreichen, wenn eine Maschine für die lange Aufgabe reserviert wird.

Werden andererseits zuerst die kurzen Aufgaben gleichmäßig über die m Maschinen verteilt und folgt dann die lange Aufgabe, so erhalten wir den Makespan $2 \cdot m - 1$.

Aufgabe 58
Wir betrachten den Fall von $m = 2$ Maschinen.

(a) Zeige, dass der oben beschriebene Greedy Algorithmus den Wettbewerbsfaktor 3/2 besitzt.

(b) Zeige, dass jede on-line Strategie mindestens den Wettbewerbsfaktor 3/2 besitzt.

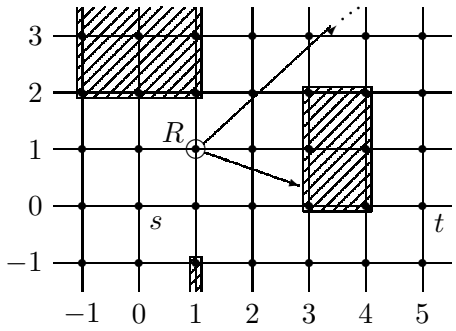
(c) Entwickle eine randomisierte on-line Strategie mit einem Wettbewerbsfaktor von höchstens 4/3.

Offenes Problem 2
Bestimme den Wettbewerbsfaktor einer besten on-line Strategie. Die gegenwärtig beste On-line strategie besitzt den Wettbewerbsfaktor 1.923.

Der Wettbewerbsfaktor einer besten randomisierten on-line Strategie ist ebenfalls nicht bekannt.

5.3 Kurze Wege in unbekanntem Terrain*

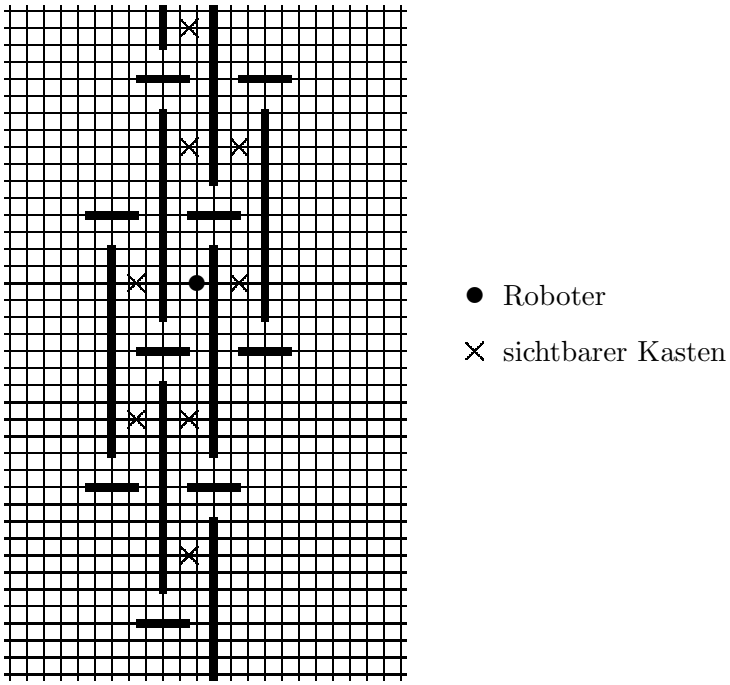
Wir platzieren einen Roboter auf dem zweidimensionalen Gitter $\mathbb{Z} \times \mathbb{Z}$. Der Roboter sitzt anfänglich auf dem Gitterpunkt $s = (0,0)$ und möchte den ihm bekannten Gitterpunkt $t = (n,0)$ erreichen. Auf dem Weg von s nach t kann der Roboter jeweils zu einem der benachbarten Gitterpunkte wechseln. Auf dem Gitter sind aber achsenparallele Rechtecke als undurchdringliche Hindernisse verteilt. Wir setzen allerdings voraus, dass keine zwei Hindernisse aneinanderstossen. Der Roboter hat keine a priori Information über die Lage der Hindernisse.



Was kann der Roboter zu einem gegebenen Zeitpunkt sehen? Er sieht jeden Punkt des Gitters, der durch eine hindernisfreie Gerade mit seinem aktuellen Aufenthaltsort verbunden ist. Wir erhalten ein on-line Problem, da zu jedem Zeitpunkt eventuell neue Teile der Umgebung bekannt werden. Können wir Strategien entwerfen, so dass der vom Roboter benutzte Weg von s nach t nicht viel länger als der kürzeste Weg ist? Unser erstes Ergebnis ist sehr enttäuschend:

Satz 5.7 *Zu jeder on-line Strategie A kann eine Umgebung konstruiert werden, so dass der kürzeste Weg von $s = (0,0)$ nach $t = (n,0)$ die Länge $O(n^{3/2})$ besitzt, der von A gewählte Weg jedoch die Länge $\Omega(n^2)$ hat.*

Beweis: Der Roboter wird auf den Gitterpunkt $s = (0,0)$ gesetzt. Wir werden den Roboter stets mit (geöffneten) Kästen umschließen. Dabei ordnen wir die Kästen so an, dass der Roboter nur seinen gegenwärtigen und wenige Nachbarkästen sehen kann. Die Kästen haben die Höhe n und Breite 2. Die anfänglich platzierten Kästen sind in der folgenden Abbildung dargestellt.



Immer wenn der Roboter in einen neuen Kasten hineinläuft, fügen wir Vertikale ein, damit die Situation des Roboters identisch zur verlassenen Situation ist. Der Roboter muss mindestens

$\frac{n}{2}$ Schritte laufen, bevor er seinen Kasten verlassen kann. Danach kann er maximal zwei horizontale Schritte laufen. Nachdem der Roboter durch $\frac{n}{4}$ Kästen laufen musste, beenden wir das Platzieren der Vertikalen und ihrer Trennwände, so dass der Roboter sein Ziel erreichen kann. Insgesamt benötigt der Roboter mindestens $\frac{n}{2} \cdot \frac{n}{4} = \Omega(n^2)$ Schritte.

Die Hindernisse haben eine Gesamtlänge von $O(n^2)$, d.h. es muss eine Zeile y_0 mit $y_0 \leq n^{3/2}$ geben, die von höchstens $O(\sqrt{n})$ Hindernissen getroffen wird. Eine off-line Strategie lässt den Roboter erst zur Koordinate $(0, y_0)$ laufen und dann die $O(\sqrt{n})$ Hindernisse mit jeweils $O(n)$ Schritten umlaufen. Danach kann der Roboter das Ziel hindernisfrei erreichen. Die Gesamtlänge des Weges ist durch $O(n^{3/2})$ beschränkt. □

Aufgabe 59

Wir betrachten das Roboterproblem, also die on-line-Version des kürzesten-Weg-Problems auf einem Gitter mit rechteckigen Hindernissen, wobei von Koordinate $(0, 0)$ die Wand $(n, i)_{i \in \mathbb{Z}}$ erreicht werden soll.

Algorithmus 5.8 bewegt den Roboter immer innerhalb eines Streifens von $-\frac{w}{2}$ bis $+\frac{w}{2}$ um die x-Achse, wobei die Streifenbreite $w \in \{n, 2n, 2^2n, 2^3n, \ldots\}$ im Laufe der Zeit eventuell vergrößert wird. Die Zahl $\tau = \frac{w}{\sqrt{n}}$ heißt Schwellenwert, SD $\in \{\text{Nord}, \text{Süd}\}$ heiße Sweep-Direction und SC $\in \{0, \ldots, \sqrt{n}\}$ heiße Sweep-Counter.

Algorithmus 5.8 (1) SD = Süd, SC = 0, $w = n$, $\tau = \frac{w}{\sqrt{n}}$.

(2) Laufe nach Osten bis zum nächsten Hindernis (oder bis die Koordinaten (n, i) für ein beliebiges i erreicht sind).

(3) y_{Nord} und $y_{\text{Süd}}$ bezeichne die y-Koordinaten der nordwestlichen bzw. südwestlichen Ecke des Hindernisses und y die y-Koordinate des Roboters.

(3a) Falls der Abstand von y zur nördlichen (bzw. südlichen) Ecke kleiner als τ ist, so umlaufe das Hindernis in dieser Richtung und kehre auf dieselbe y-Koordinate zurück.

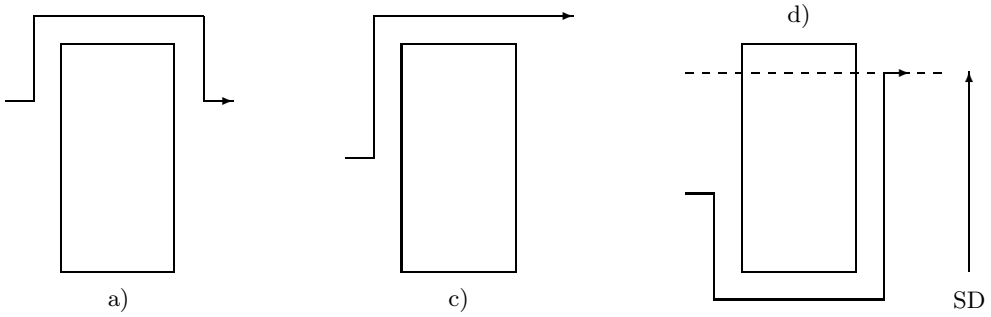
(3b) Ragen beide Ecken aus dem Streifen, d.h. $y_{\text{Nord}} > \frac{1}{2}w$ und $y_{\text{Süd}} < -\frac{1}{2}w$, dann setze $w := 2w$, $\tau := 2\tau$, SC = 0 und SD = Süd.

(3c) Beide vorige Fälle gelten nicht und $|y_{\text{SD}}| \leq \frac{1}{2}w$. Dann umlaufe das Hindernis in Richtung SD und bleibe auf y-Koordinate y_{SD}.

(3d) Die Fälle a),b) und c) treffen nicht zu, also $|y_{\text{SD}}| > \frac{1}{2}w$. Laufe in Gegenrichtung zu SD am Hindernis vorbei, dann in Richtung SD bis zum Rand des Streifens. Wechsele SD, SC := SC + 1.

(4) IF SC $> \sqrt{n}$ THEN SC = 0, SD = Süd, $w = 2w$ und $\tau = 2\tau$.

(5) Gehe zu Schritt (2).



Sei w_f die Streifenbreite am Ende. Zeige, dass die Länge des gelaufenen Wegs durch $O(w_f \cdot \sqrt{n})$ beschränkt ist, und dass die Länge des kürzesten Wegs mindestens $\Omega(w_f)$ beträgt. Damit ist der Wettbewerbsfaktor des Algorithmus' $O(\sqrt{n})$ und der Algorithmus ist asymptotisch optimal, da die untere Schranke von $\Omega(\sqrt{n})$ getroffen wird.

Wo liegt die Schwierigkeit für einen on-line Algorithmus? Die Rechteck-Hindernisse sind degeneriert und der Roboter profitiert nicht von einer langen vertikalen Wanderung durch eine

entsprechend lange horizontale Wanderung. Sei R ein Rechteck mit Seitenlängen a und b. Dann nennen wir

$$\lambda := \max\{\frac{a}{b}, \frac{b}{a}\}$$

den *Degenerierungsfaktor* des Rechtecks. Insbesondere ist der Degenerierungsfaktor genau dann 1, wenn ein Quadrat vorliegt.

Satz 5.9 *Es sei eine Umgebung mit Startpunkt $s = (0, 0)$ und Ziel $t = (n, 0)$ gegeben, die nur aus Rechteck-Hindernissen mit Degenerierungsfaktor höchstens λ bestehe.*

(a) Dann gibt es eine on-line Strategie mit Wettbewerbsfaktor höchstens $\frac{\lambda}{2} + 1$.

(b) Für $\lambda = O(\sqrt{n})$ besitzt jede on-line Strategie einen Wettbewerbsfaktor von $\Omega(\lambda)$.

Beweis (a): Wir geben eine einfache on-line Strategie an:

(1) Der Roboter beginnt auf der Linie $(i, 0)$ und bewegt sich in horizontaler Richtung solange nach rechts wie dies möglich ist.

(2) Wenn der Roboter auf ein Hindernis trifft, umläuft er dieses Hindernis mit optimalen Weg, um wieder auf y-Koordinate 0 zu gelangen.

Es werden höchstens $n \cdot \frac{\lambda}{2} + n$ Schritte gemacht. Da der optimale Weg mindestens n Schritte lang ist, folgt die Behauptung.

(b) Wir verfahren wie in Satz 5.7. Diesmal können wir aber nur erzwingen, dass nach zwei horizontalen Schritten mindestens $\frac{\lambda}{2}$ vertikale Schritte benötigt werden und erhalten die Mindestlänge $(\frac{\lambda}{2} + 2) \cdot \frac{n}{2} = \frac{\lambda+4}{4} \cdot n$.

Wie sieht ein kürzester Weg aus? Ein Hinderniss taucht in höchstens λ y-Koordinaten auf, d.h. es gibt eine Zeile y_0 mit $|y_0| \leq \lambda\sqrt{n}$, in der nur $O(\sqrt{n})$ Hindernisse vorkommen. Dies führt auf einen Weg der Länge $n + O(\lambda\sqrt{n})$ und das Ergebnis folgt für $\lambda = O(\sqrt{n})$. □

Kapitel 6

Das Paging-Problem

Angenommen, wir können nur wenige Seiten eines langsamen externen Speichers in einem schnellen internen Speicher halten. Wenn eine neue Seite angefordert wird, müssen wir eine andere Seite auslagern. Wie bestimmen wir die auszulagernde Seite?

Zur Formalisierung nehmen wir an, dass genau k Seiten intern gespeichert werden können. Ein on-line Algorithmus für das Paging-Problem erhält für jede angeforderte, aber nicht vorhandene Seite einen Strafpunkt. Welche Wettbewerbsfaktoren können erreicht werden?

6.1 Deterministische Strategien

Wir betrachten die folgenden On-Line Strategien.

- LFU (Least-Frequently-Used): Lagere die am wenigsten angeforderte Seite aus. Dabei zählen wir die Anzahl der Anforderungen vom Zeitpunkt der letzten Einlagerung der Seite an.

- LRU (Least-Recently-Used): Lagere die Seite aus, deren letzte Anforderung am weitesten zurückliegt.

- FIFO (First-In-First-Out): Lagere die Seite aus, die am längsten gespeichert wurde.

- RANDOM: Verdränge eine zufällig und uniform gewählte Seite aus dem Speicher.

Aufgabe 60
Die RANDOM Strategie für das Paging-Problem verdrängt für eine neu zu ladende Seite eine zufällige Seite aus dem Speicher. Zeige, dass die RANDOM Strategie einen Wettbewerbsfaktor von mindestens k besitzt. Hinweis: Seien $a_1, a_2, \ldots, a_k, b_1, b_2, \ldots$ voneinander verschiedene Seiten. Betrachte eine Folge

$$a_1, a_2, \ldots, a_k, (b_1, a_2, \ldots, a_k)^1, (b_2, a_2, \ldots, a_k)^2, (b_3, a_2, \ldots, a_k)^3, \ldots$$

wobei $(s)^l$ eine l-fache Wiederholung der Folge s bedeutet.

Wir untersuchen zuerst die LFU Strategie. Bei einer Speicherfähigkeit von k Seiten werden zuerst die Seiten $p_1, p_2, \ldots, p_k$ nacheinander eingelesen, gefolgt von der Anforderungsfolge $p_2, \ldots, p_k$. Danach starten wir die Anforderungen $p_0, p_1, p_0, p_1, \ldots$, die jeweils zur Auslagerung der anderen Seite führen. Der Wettbewerbsfaktor ist durch keine Konstante beschränkt, denn eine optimale off-line Strategie erhält nur einen Strafpunkt.

Aufgabe 61
Die Paging Strategie Flush-when-Full leert den Seitenspeicher vollständig, wenn ein Seitenfehler auftritt und kein Platz mehr für die neue Seite vorhanden ist. Daraufhin wird die neue Seite eingelagert.

Bestimme den Wettbewerbsfaktor der Strategie.

Satz 6.1 *Jeder deterministische on-line Algorithmus für das Paging-Problem mit k Seiten hat mindestens den Wettbewerbsfaktor k.*

Beweis: Sei A ein on-line Algorithmus und σ eine Folge, die zu einem ersten Strafpunkt führt. Zu diesem Zeitpunkt habe A die Seiten $(p_1, \ldots, p_k)$ gespeichert und die Seite p_0 ausgelagert. Da A deterministisch arbeitet, lässt sich eine Eingabefolge σ' (mit Anforderungen an die $k+1$ Seiten $p_0, \ldots, p_k$) konstruieren, so dass A einen Fehler auf jeder neu angeforderten Seite macht.

Eine optimale off-line Strategie erhält andererseits auf $\sigma\sigma'$ (mit Anforderungen an nur $k+1$ Seiten) höchstens $\frac{|\sigma\sigma'|}{k}$ Strafpunkte. (Warum?) Damit ist Satz 6.1 bewiesen. □

FIFO und LRU sind optimale Paging Strategien, denn sie erreichen den Wettbewerbsfaktor k.

Satz 6.2 *Die Strategien LRU und FIFO besitzen den Wettbewerbsfaktor k.*

Beweis Sei $\sigma = (\sigma_1, \sigma_2, \ldots)$ eine Eingabefolge, auf der LRU am schlechtesten abschneidet. Wir zerlegen σ in Teilfolgen $\sigma = (\sigma^1, \sigma^2, \ldots)$, so dass

- σ^1 mit dem ersten Strafpunkt von LRU endet und
- σ^j für $j \geq 2$ nach Strafpunkt $(j-1) \cdot k + 1$ endet.

Es genügt zu zeigen, dass jede off-line Strategie während jeder Teilfolge mindestens einen Strafpunkt erhält. Dies ist offenbar richtig für σ^1, da LRU mit anfänglich leerem Speicher nur einen Strafpunkt nach $k+1$ Anforderungen erhält. Betrachten wir σ^j für $j \geq 2$. Wir unterscheiden die folgenden Fälle:

Fall 1: LRU erhält in σ^j zwei Strafpunkte für dieselbe Seite p. Nach dem ersten Strafpunkt für p wird p in den Speicher geholt. Da ein zweiter Strafpunkt vergeben wird, wurde p zwischenzeitlich ausgelagert. Nach Definition von LRU kann dies nur passieren, wenn zwischen dem ersten und zweiten Strafpunkt k neue Seiten angefordert werden. Insgesamt fordert σ^j mindestens $k+1$ *verschiedene* Seiten an, und damit muss jede off-line Strategie auch mindestens einen Strafpunkt erhalten.

Fall 2: LRU erhält in σ^j Strafpunkte für k verschiedene Seiten. Sei q die Seite, auf der LRU den letzten Strafpunkt in σ^{j-1} erhält. Wir unterscheiden zwei Fälle:

Fall 2.1 : LRU erhält einen Strafpunkt für q in σ^j. Dieser Fall verläuft wie Fall 1.

Fall 2.2 : LRU erhält keinen Strafpunkt für q in σ^j. Eine optimale off-line Strategie muss zu Beginn von σ^j die Seite q speichern, da gerade eine Anforderung für q erfolgte. Nach Annahme werden k verschiedene Seiten in σ^j angefordert, die alle von q verschieden sind. Damit muss jede off-line Strategie irgendwann einen Strafpunkt in σ^j erhalten.

Die Behauptung für FIFO folgt analog. □

6.2 Randomisierte Strategien

Für randomisierte on-line Algorithmen erhalten wir einen besseren Wettbewerbsfaktor mit der Marking Strategie.

Algorithmus 6.3 **Die Marking Strategie**

(1) Die Seite p werde angefordert.

(2) Wenn p nicht gespeichert ist, wähle zufällig eine nicht-markierte Seite und lagere sie aus.

Wenn alle Seiten markiert sind, lösche die Markierungen und wähle zufällig eine der unmarkierten Seiten zur Auslagerung.

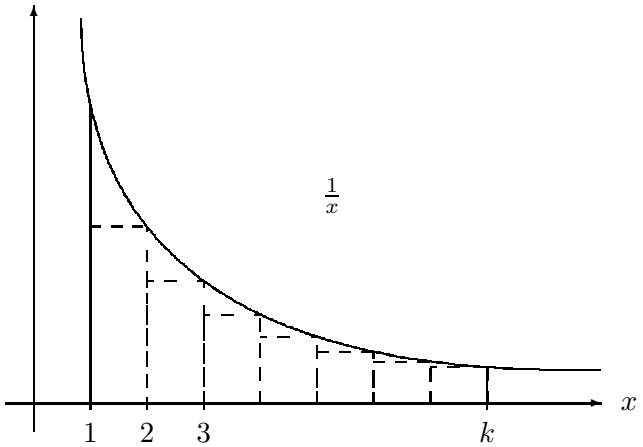
(3) Markiere p.

Wann ist der Entwurf randomisierter on-line Algorithmen vielversprechend? Wenn wir eine Sammlung deterministischer Algorithmen haben, so dass *jede* Eingabe von den *meisten* Algorithmen dieser Sammlung mit geringen Kosten abgearbeitet wird: In einem solchen Fall wählen wir randomisiert einen Algorithmus dieser Sammlung zu Anfang der Berechnung und haben die berechtigte Hoffnung, dass die Wahl für die unbekannte Eingabefolge kostengünstig ist.

Die Marking Strategie versucht, häufig nachgefragte Seiten im Speicher zu halten, da diese Seiten sofort markiert werden. Diese Vorgehensweise basiert auf der guten Strategie LRU, mit der zufälligen Wahl einer auszulagernden Seite erhalten wir viele LRU-Varianten, von denen hoffentlich die meisten nur geringe Kosten verursachen. Allerdings könnten wir Marking auch als eine randomsierte Variante der schlechten deterministischen LFU Strategie ansehen und das verheißt nichts Gutes.

Satz 6.4 *Die Marking Strategie besitzt höchstens den Wettbewerbsfaktor* $2 \cdot \sum_{i=1}^{k} \frac{1}{i}$.

Wir wenden das Integralkriterium an



und erhalten

$$\sum_{i=1}^{k} \frac{1}{i} \leq 1 + \int_{1}^{k} \frac{1}{x} dx = \ln(k) + 1,$$

d.h. wir haben den besten Wettbewerbsfaktor für deterministische Strategien wesentlich von k auf $2\ln(k)$ verbessern können.

Beweis von Satz 6.4: Für eine beliebige Eingabefolge σ müssen wir zeigen, dass der Erwartungswert $E[f(\sigma, A(\sigma))]$ im Vergleich zu $f(\sigma, \mathrm{Opt}(\sigma))$ nicht zu groß wird. Dazu zerlegen wir σ in Teilfolgen $\sigma = (\sigma^0, \sigma^1, \ldots)$, so dass σ^1 mit dem ersten vergebenen Strafpunkt beginnt. Weiter sei σ^i für $i \geq 1$ ein längstes auf σ^{i-1} folgendes Eingabeintervall, in dem genau k verschiedene Seiten angefordert werden. S_i sei die Menge der vor Beginn von σ^i gespeicherten Seiten. Wir beginnen mit einigen fundamentalen Beobachtungen:

Behauptung 6.1 *Am Anfang von σ^i sind für $i \geq 1$ alle gespeicherten Seiten markiert. Genau diejenigen Seiten sind gespeichert, die während σ^{i-1} angefordert werden.*

Beweis: Durch Induktion über i. Zu Beginn von σ^1 erhält Marking einen Strafpunkt. Zu diesem Zeitpunkt hat Marking die k verschiedenen, während σ^0 angeforderten Seiten gespeichert und markiert. Diese Markierungen werden gelöscht. Während des Abarbeitens von σ^1 angeforderte Seiten werden sofort markiert *und* diese Markierung wird während der Abarbeitung von σ^1 nicht gelöscht, da es noch unmarkierte Seiten gibt. Am Ende von σ^1 sind somit genau die während σ^1 angeforderten Seiten markiert. Der Induktionsschritt ist identisch. □

Behauptung 6.2 *Die Zerlegung von σ wie auch die Mengen S_i sind unabhängig von den Münzwürfen von Marking.*

Beweis: Dies ist eine unmittelbare Konsequenz aus der Definition der Zerlegung und Behauptung 6.1. □

Um die Überlegungen abzuschließen, unterscheiden wir die während σ^i ($i \geq 1$) ausgeführten Anforderungen von k verschiedenen Seiten in

- *neue* Anforderungen, wenn die angeforderte Seite nicht in S_i liegt und
- *alte* Anforderungen, wenn die angeforderte Seite in S_i liegt.

Sei n_i die Anzahl der neuen Anforderungen während der Abarbeitung von σ^i. Die Behauptung des Satzes folgt, wenn wir die nächsten beiden Behauptungen nachweisen.

Behauptung 6.3 *Die erwartete Anzahl von Strafpunkten für Marking während der Abarbeitung von σ^i ist höchstens*

$$n_i \cdot \sum_{j=1}^{k} \frac{1}{j}.$$

Behauptung 6.4 *Jeder off-line Algorithmus erhält nach Abarbeitung von $\sigma = (\sigma^0, \ldots, \sigma^m)$ mindestens*

$$\frac{1}{2} \cdot \sum_{i=1}^{m} n_i$$

Strafpunkte.

Die Behauptung des Satzes folgt, da Marking höchstens die erwartete Anzahl von

$$\sum_{i=1}^{m} n_i \cdot \sum_{j=1}^{k} \frac{1}{j}$$

Strafpunkte erhält. Beachte, dass Marking während σ^0 keinen Strafpunkt erhält.

Beweis von Behauptung 6.3: Wir betrachten zuerst die erwartete Anzahl $\text{strafe}^i_{\text{alt}}$ von Strafpunkten für die $k - n_i$ alten Anforderungen. Wir beachten zuerst, dass $\text{strafe}^i_{\text{alt}}$ maximal ist, wenn die Anforderungen nach neuen Seiten den Anforderungen nach alten Seiten vorangehen, da dann die Wahrscheinlichkeit maximiert wird, dass eine nachgefragte alte Seite zwischenzeitlich ausgelagert wurde.

Sei alt_1 die als erste angeforderte alte Seite. Die Anforderung nach alt_1 führt mit Wahrscheinlichkeit höchstens $\frac{n_i}{k}$ auf einen Strafpunkt: Die Gegenwahrscheinlichkeit, also die Wahrscheinlichkeit in den notwendigen n_i Auslagerungen *nicht* beteiligt zu sein, beträgt $\binom{k-1}{n_i}/\binom{k}{n_i} = (k - n_i)/k = 1 - n_i/k$.

Sei alt_2 die als zweite angeforderte alte Seite. Die Wahrscheinlichkeit eines Strafpunkts für alt_2 verändert sich auf höchstens $\frac{n_i}{k-1}$. Warum? Wir betrachten zuerst den Fall, dass die Anforderung nach alt_1 nicht auf einen Strafpunkt geführt hat. Die Seite alt_1 wurde also nicht ausgelagert, um Platz für neue Seiten zu schaffen, und dementsprechend „sollte“ die Wahrscheinlichkeit einer Auslagerung von alt_2 leicht ansteigen. Tatsächlich ist die Menge der für eine Auslagerung in Frage kommenden Kandidaten auf $k - 1$ geschrumpft und die Wahrscheinlichkeit eines Strafpunkts für alt_2 steigt damit auf $n_i/(k-1)$. Wurde alt_1 hingegen ausgelagert, wird die Wahrscheinlichkeit einer Auslagerung von alt_2 eher leicht abnehmen, auf jeden Fall aber durch n_i/k nach oben beschränkt bleiben. Natürlich muss unsere Analyse den schlechteren der beiden Fälle annehmen, und wir erhalten $n_i/(k-1)$ als obere Schranke der Wahrscheinlichkeit eines Strafpunkts für Seite alt_2.

Wenn wir das Argument wiederholen, werden wir auf die Abschätzung

$$\text{strafe}^i_{\text{alt}} \leq \frac{n_i}{k} + \frac{n_i}{k-1} + \cdots + \frac{n_i}{k-(k-n_i-1)}$$

geführt. Die erwartete Anzahl $\text{strafe}^i_{\text{neu}}$ aller während σ^i ausgeführten neuen Anforderungen ist n_i, so dass

$$\text{strafe}^i_{\text{alt}} + \text{strafe}^i_{\text{neu}} \leq n_i + n_i \cdot \left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{n_i+1}\right) \leq n_i \cdot \sum_{j=1}^{k} \frac{1}{j}.$$

folgt. Damit ist Behauptung 6.3 gezeigt. □

Beweis von Behauptung 6.4: Sei A ein off-line Algorithmus. Wir analysieren die Strafpunktzahl von A mit Hilfe der Potentialfunktion Φ, wobei

$$\Phi(i) = \begin{cases} \text{Anzahl der Seiten, die } \textit{von } A\textit{, aber nicht} \\ \textit{von Marking} \text{ zu Beginn der Abarbeitung} \\ \text{von } \sigma^i \text{ gespeichert sind} \end{cases}$$

Sei $\text{strafe}^i(A)$ die von A während σ^i erhaltene Anzahl von Strafpunkten. Dann ist

$$\text{strafe}^i(A) \geq n_i - \Phi(i),$$

denn höchstens $\Phi(i)$ angeforderte neue Seiten werden ohne Strafpunkt abgearbeitet.

Nach Abarbeiten von σ^i besitzt A genau $\Phi(i+1)$ Seiten, die Marking nicht besitzt und umgekehrt. Marking speichert diese $\Phi(i+1)$ Seiten nur, wenn sie während σ^i angefordert

werden. Da A diese Seiten nicht besitzt, hat A die Seiten während σ^i ausgelagert und jeweils einen Strafpunkt erhalten. Damit gilt

$$\text{strafe}^i(A) \geq \Phi(i+1).$$

Da allgemein $\max\{a,b\} \geq \frac{1}{2}(a+b)$ gilt, erhalten wir

$$\text{strafe}^i(A) \geq \frac{1}{2}\big(n_i - \Phi(i) + \Phi(i+1)\big)$$

und

$$\begin{aligned}\sum_{i=1}^{m} \text{strafe}^i(A) &\geq \frac{1}{2}\cdot\sum_{i=1}^{m}\big(n_i - \Phi(i) + \Phi(i+1)\big)\\ &= \left(\frac{1}{2}\cdot\sum_{i=1}^{m} n_i\right) + \frac{1}{2}\big(\Phi(m+1) - \Phi(1)\big)\\ &\geq \frac{1}{2}\cdot\sum_{i=1}^{m} n_i.\end{aligned}$$

Dies beweist Behauptung 6.4 □

und damit ist Satz 6.4 gezeigt. □

Aufgabe 62
Zeige für den Fall $k = 2$, dass der Marking Algorithmus einen Wettbewerbsfaktor von mindestens $(2\sum_{i=1}^{k}\frac{1}{i})-1$ hat.

Unser nächstes Ziel ist der Nachweis, dass Marking fast optimal ist, also einen fast minimalen Wettbewerbsfaktor besitzt. Wie zeigt man aber, dass randomisierte on-line Strategien keine sehr viel kleineren Wettbewerbsfaktoren besitzen können? Ziel unserer allgemeinen Vorgehensweise ist eine Reduktion der Analyse randomisierter Strategien auf die Analyse deterministischer Strategien.

Dazu beachte, dass wir eine randomisierte Stratetgie R als eine Sammlung deterministischer Strategien $(D_r \mid r)$ auffassen können: jeweils eine deterministische Strategie D_r für eine Folge r von Münzwürfen der randomisierten Strategie R. Für Paging wie auch allgemein ist das folgende Vorgehen erfolgreich:

(1) Bestimme eine Verteilung π auf den Eingabefolgen σ.

(2) Bestimme eine möglichst gute untere Schranke für die *erwartetete* Strafpunktzahl

$$E_\pi[D] = \sum_\sigma \text{prob}_\pi[\sigma]\cdot\text{Strafe}(D(\sigma))$$

einer beliebigen deterministischen on-line Strategie D und vergleiche $E_\pi[D]$ mit der erwarteten Strafpunktzahl

$$E_\pi[\text{Opt}] = \sum_\sigma \text{prob}_\pi[\sigma]\cdot\text{Strafe}(\text{Opt}(\sigma))$$

einer optimalen Strategie.

Natürlich können wir auch die erwartete Strafpunktzahl der randomisierten Strategie R betrachten, müssen diesmal aber auch die internen Münzwürfe von R berücksichtigen: Wenn $R = (D_r \mid r)$ und die deterministische Strategie D_r mit Wahrscheinlichkeit $p[r]$ ausgeführt wird, dann werden wir auf die Definition

$$E_\pi[R] := \sum_\sigma \text{prob}_\pi[\sigma] \cdot (\sum_r p[r] \cdot \text{Strafe}(D_r(\sigma)))$$

geführt und es gilt nach Summenvertauschung

$$E_\pi[R] = \sum_r p[r] \cdot \sum_\sigma \text{prob}_\pi[\sigma] \cdot \text{Strafe}(D_r(\sigma)) = \sum_r p[r] \cdot E_\pi[D_r]. \tag{6.1}$$

Angenommen, wir können zeigen, dass die erwartete Strafpunktzahl einer *jeden* deterministischen Strategie sehr viel größer als die erwartete Strafpunktzahl einer optimalen Strategie ist, dass also

$$E_\pi[D] \geq \alpha \cdot E_\pi[\text{Opt}]$$

für jede deterministische Strategie D gilt. Dann erhalten wir

$$E_\pi[R] = \sum_r p[r] \cdot E_\pi[D_r] \geq \alpha \cdot \sum_r p[r] \cdot E_\pi[\text{Opt}] = \alpha \cdot E_\pi[\text{Opt}]$$

als Konsequenz von (6.1). Wenn aber $E_\pi[R] \geq \alpha \cdot E_\pi[\text{Opt}]$ als Ungleichung zwischen Erwartungswerten gilt, dann muss es eine Folge σ mit $\sum_r p[r] \cdot \text{Strafe}(D_r(\sigma)) \geq \alpha \cdot \text{Strafe}(\text{Opt}(\sigma))$ geben und der Wettbewerbsfaktor von R ist damit mindestens α. Wir fassen zusammen:

Lemma 6.5 *Wenn $E_\pi[D] \geq \alpha \cdot E_\pi[\text{Opt}]$ für jede deterministische Strategie D gilt, dann besitzt jede randomisierte on-line Strategie für das Paging Problem mindestens den Wettbewerbsfaktor α.*

Der zentrale Schritt in diesem Vorgehen ist damit die Konstruktion einer schwierigen Verteilung π, also einer Verteilung, die die erwartete Strafpunktzahl deterministischer on-line Strategien möglichst hoch treibt, aber gleichzeitig die erwartete Strafpunktzahl einer optimalen Strategie nicht zu hoch treibt. Für das Paging Problem ist die Konstruktion von π einfach.

Satz 6.6 *Jeder randomisierte on-line Algorithmus für das Paging-Problem mit k Seiten besitzt einen Wettbewerbsfaktor $\alpha \geq \sum_{i=1}^{k+1} \frac{1}{i}$.*

Beweis: Wir definieren zuerst die Verteilung π. Nur solche Folgen σ erhalten eine positive Wahrscheinlichkeit, die genau $k \cdot (k+1)$ Anfragen stellen und jedesmal nur Seiten aus $\{0, 1, \ldots, k\}$ nachfragen. Alle Folgen mit positiver Wahrscheinlichkeit sind gleichwahrscheinlich.

Unter der Annahme, dass zu Anfang irgendwelche k dieser $k+1$ Seiten gespeichert sind, ist die Wahrscheinlichkeit eines Strafpunkts genau $\frac{1}{k+1}$ und wir erhalten

$$E_\pi[D] \geq \frac{k \cdot (k+1)}{k+1} = k$$

für die erwartete Strafpunktzahl $E_\pi[D]$ einer deterministischer Strategie D. Es bleibt zu zeigen, dass die erwartete Strafpunktzahl einer besten off-line Strategie höchstens

$$\frac{k}{\sum_{i=1}^{k+1} \frac{1}{i}}$$

beträgt. Sei σ eine zufällig gewählte Folge, die nur die Seiten $0, 1, \ldots, k$ nachfragt. Wir unterteilen $\sigma = (\sigma^0, \sigma^1, \ldots)$ in Teilfolgen, so dass Teilfolge σ^j längstmöglich unter der Einschränkung gewählt wird, dass nur Anforderungen an k verschiedene Seiten gemacht werden. Eine optimale off-line Strategie wird in jeder Teilfolge σ^j höchstens einen Strafpunkt erhalten und zwar für die zu Beginn von σ^j nicht gespeicherte Seite. Die erwartete Anzahl von Strafpunkten für σ ist somit durch die erwartete Anzahl der Teilfolgen von σ beschränkt. Wieviele Teilfolgen sind zu erwarten? Antwort:

$$\frac{\text{Folgenlänge}}{\text{erwartete Länge einer Teilfolge}}.$$

Die erwartete Länge einer Teilfolge ist die durchschnittliche Wartezeit, bis wir $k+1$ verschiedene aus $k+1$ möglichen Seiten „gezogen“ haben.

Fakt 6.1 *Die durchschnittliche Wartezeit bis n Objekte aus der Menge $\{1, \ldots, n\}$ gezogen wird, beträgt $n \cdot \sum_{i=1}^{n} \frac{1}{i}$.*

Die erwartete Anzahl von Strafpunkten einer optimalen Strategie für σ ist also $\frac{k \cdot (k+1)}{(k+1) \cdot \sum_{i=1}^{k+1} \frac{1}{i}} = \frac{k}{\sum_{i=1}^{k+1} \frac{1}{i}}$ und das war zu zeigen. □

Kapitel 7

Das k-Server-Problem*

Wir führen das k-Server-Problem ein, das sich als eine weitreichende Verallgemeinerung des Paging-Problem herausstellen wird. Zur Vorbereitung benötigen wir das Konzept eines metrischen Raumes.

Definition 7.1
Ein metrischer Raum ist ein Paar (X, d) bestehend aus einer Menge X und einer Funktion $d : X \times X \to \mathbb{R}$. Die Funktion d besitzt die Eigenschaft einer Metrik, d.h. es gilt

(a) Definitheit: $d(x, y) = 0 \Leftrightarrow x = y$ für alle $x, y \in X$.

(b) Symmetrie: $d(x, y) = d(y, x)$ für alle $x, y \in X$.

(c) Dreiecksungleichung: $d(x, y) + d(y, z) \geq d(x, z)$ für alle $x, y, z \in X$.

Für jeden metrischen Raum (X, d) gilt $d(x, y) \geq 0$ für alle $x, y \in X$, denn $0 = d(x, x) \leq d(x, y) + d(y, x) = 2d(x, y)$.

Beispiel 7.1 Für $X = \mathbb{R}^n$ sind

- $d_1(x, y) = ||x - y||_1 = \sum_{i=1}^{n} |x_i - y_i|$,
- $d_2(x, y) = ||x - y||_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$,
- $d_p(x, y) = ||x - y||_p = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}$ für $p \in \mathbb{N}$ und
- $d_\infty = \max\{|x_i - y_i| \;\; 1 \leq i \leq n\}$

Metriken. Ebenso ist die Funktion $d_{\text{Gleichheit}} : X \times X \to \{0, 1\}$ mit

$$d_{\text{Gleichheit}}(x, y) = \begin{cases} 0 & \text{falls } x = y \\ 1 & \text{sonst} \end{cases}$$

eine Metrik.

Definition 7.2 Sei (X, d) ein metrischer Raum zu einer endlichen Menge X. Weiterhin sei eine Menge von k Servern gegeben. Eine Eingabefolge für das k-Server-Problem besteht aus einer Folge σ von Elementen in X. Für $\sigma = (\sigma_1, \sigma_2, \ldots)$ muss nach Abarbeiten der Teilfolge

$(\sigma_1, \ldots, \sigma_{i-1})$ einer der k Server der Anforderung σ_i nachkommen und von seinem gegenwärtigen Aufenthaltsort nach σ_i wandern. Sei

$$f_\sigma(\tau) = \left\{ \begin{array}{l} \text{die von } \tau \text{ induzierte Länge} \\ \text{aller Serverbewegungen} \end{array} \right.$$

Die Ausgabefolge τ ist die Folge aller Serverbewegungen.

Bevor wir zu Anwendungen des Server-Problems kommen, geben wir eine Normalform optimaler on-line Strategien an.

Lemma 7.3 *Während der Abarbeitung einer einzelnen Anforderung genügt es, genau einen Server zu bewegen.*

Beweis: Angenommen, es werden zwei Server während einer einzelnen Anforderung bewegt. Erfüllt der zusätzlich bewegte Server später selbst eine Anforderung, hätten wir ihn später von seinem ursprünglichen Aufenthaltsort losschicken können, ohne die Weglänge zu vergrößern (Dreiecksungleichung der Metrik). □

Beispiel 7.2 **Anwendungen des Server-Problems**

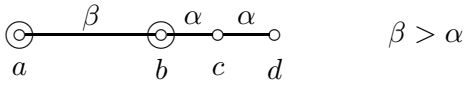
- Das Paging-Problem ist ein Spezialfall, wenn wir X als die Menge der Seiten und $d = d_{\text{Gleichheit}}$ wählen.
- Wir möchten zwei Köpfe einer Festplatte so über die Spuren der Platte bewegen, dass die insgesamt zurückgelegte Strecke zur Erfüllung aller I/O-Anfragen minimal ist. Wir wählen X als die Menge der Spuren und $d = d_1$.
- Eine Taxizentrale versucht, ihre Fahrer so zu positionieren, dass Anforderungen schnellst möglich erfüllt werden. Dazu zerlegen wir das Stadtgebiet in Stadtteile und modellieren die Stadtteile als Knoten eines gerichteten Graphen. Eine gerichtete Kante von Stadtteil u nach Stadtteil v hat als Kantengewicht die Zeit für die Fahrt von u nach v, falls v ein benachbarter Stadtteil von u ist, und ansonsten das Gewicht ∞. Wir erhalten eine Metrik $d : X \times X \to \mathbb{R} \cup \{\infty\}$ auf der Menge X aller Stadtteile, wenn wir $d(u, v)$ als die Länge eines kürzesten Weges von u nach v definieren.

Um mit dem Server-Problem vertraut zu werden, versuchen wir einige naheliegende Ansätze.

***Der Greedy-Ansatz*:** Erfülle eine Anforderung durch den nächstliegenden Server.

Diese Strategie scheint sinnvoll, ist allerdings nicht erfolgreich wie wir gleich sehen werden. Darüberhinaus ist die Strategie nicht vollständig definiert, da nicht entschieden wird, welcher von zwei Servern mit gleichem Abstand zu wählen ist. (In der Anwendung auf das Paging Problem ist demgemäß irgendeine Seite auszulagern, wenn eine Anforderung für eine nicht gespeicherte Seite erfolgt.)

Der Greedy-Ansatz verhält sich beispielsweise katastrophal für die Menge $X = \{a, b, c, d\}$ und zwei Servern, die anfänglich auf a und b stehen.



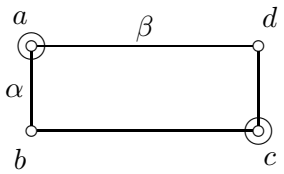
$\beta > \alpha$

Eingabefolgen aus $\{c,d\}^*$ bewegen stets nur den ursprünglich auf b sitzenden Server. Da es genügt, die beiden Server einmalig auf c und d zu setzen, gibt es keine Konstante γ, so dass der Greedy-Ansatz den Wettbewerbsfaktor γ erreicht.

Die Balance Strategie schneidet bei diesem Beispiel wesentlich besser ab. Jeder Server i merkt sich die Gesamtlänge L_i seiner bisher gelaufenen Strecke. Bei einer Anforderung für Position x berechnet Server i mit Aufenthaltsort x_i die Summe

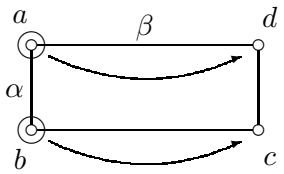
$$\text{Last}_i := L_i + d(x_i, x)$$

und der Server mit kleinstem Last-Wert kommt der Anforderung nach. Wir konstruieren dennoch eine Eingabefolge mit großem Wettbewerbsfaktor.



$\beta \gg \alpha$

Betrachte das Server Problem zur obigen Abbildung und wähle die Eingabefolge $(abcd)^n$. Ein optimaler off-line-Algorithmus platziert seine beiden Server auf a und c und läuft für jede Teilfolge $abcd$ eine Distanz von $\leq 4\alpha$ (einschließlich des Platzierens der Server auf a und c). Angenommen, wir platzieren die beiden Server für die Balance Strategie anfänglich auf a und b.



$\beta \gg \alpha$

Die beiden Server wandern die Streckenlänge 2β. Bei der nächsten Anforderungsfolge $abcd$ wird sogar die Streckenlänge 4β gelaufen: Es gibt keine Konstante γ, so dass die Balance Strategie den Wettbewerbsfaktor γ erreicht.

Erstaunlicherweise wird die Balance Strategie wesentlich besser, wenn die Summe $L_i + d(x, x_i)$ durch $L_i + 2d(x, x_i)$ ersetzt wird: Für $k = 2$ erhält man den Wettbewerbsfaktor 10 (siehe [IR]).

Die randomisierte Greedy-Methode ist ebenfalls erfolgreicher. Dabei wird ein Server zufällig gewählt, wobei die Wahrscheinlichkeit p_i, dass Server i mit Standort x_i bei Anforderung x gewählt wird, invers proportional zur Distanz $d(x, x_i)$ ist, also

$$p_i = \frac{1}{d(x,x_i)} \cdot \frac{1}{\sum_{j=1}^{k} \frac{1}{d(x,x_j)}}$$

gilt. E. Grove [G] zeigt, dass diese randomisierte Variante den Wettbewerbsfaktor

$$\alpha = \frac{5}{4} k 2^k - 2k$$

besitzt. Dieses Ergebnis ist nicht sehr ermutigend, obwohl sehr viel besser als für die bisherigen Methoden.

Aufgabe 63
Beim *Feuerwehrproblem* wird zunächst ein metrischer Raum vorgegeben, in dem k Punkte als Feuerwehrstationen ausgezeichnet sind. On-line werden dann nacheinander k Punkte angegeben, an welchen Brände entstanden sind. Jedem Brand soll eine Feuerwehrstation zugeordnet werden, wobei eine Feuerwehrstation nur einen Brand übernehmen kann. Ziel ist es, die Summe der Distanzen zwischen Feuerwehrstationen und zugehörigen Bränden zu minimieren. Der wesentliche Unterschied zum k-Server Problem ist also, dass die Server jeweils nur eine Aufgabe bekommen.

(a) Zeige, dass jeder on-line-Algorithmus, der das Feuerwehrproblem löst, einen Wettbewerbsfaktor von mindestens $2k - 1$ hat.

(b) Die Nearest-Neighbor-Strategie für das Feuerwehrproblem wählt immer die am nächsten zum Brand liegende Feuerwehrstation aus. Zeige, dass die Nearest-Neighbor-Strategie einen in k exponentiell schlechten Wettbewerbsfaktor hat.

Aufgabe 64
Das *Taxizentralenproblem* ist eine Umkehrung des Feuerwehrproblems. Gegeben sind ein metrischer Raum sowie k Positionen, an denen Taxen auf Kundschaft warten. Kunden müssen die (direkte) Anfahrt ihres Taxis nach Entfernung bezahlen. Die Taxizentrale versucht, k nacheinander anrufende Kunden so zu bedienen, dass maximale Kosten verursacht werden. Die Farthest-Neighbor-Strategie für das Taxizentralenproblem wählt immer das am weitesten vom Kunden wartende Taxi aus.

(a) Zeige, dass die Farthest-Neighbor-Strategie einen Wettbewerbsfaktor von höchstens 3 hat, d.h. eine Lösung bestimmt, welche mindestens $\frac{1}{3}$ der Kosten des Optimums hat.

(b) Zeige, dass jeder deterministische on-line-Algorithmus für das Taxizentralenproblem mindestens den Wettbewerbsfaktor 3 hat.

Wir beschreiben zuletzt die wesentlich erfolgreichere ***Work-Function Strategie***, die den Wettbewerbsfaktor $2k$ hat. Man vermutet, dass der tatsächliche Wettbewerbsfaktor k ist, d.h. die Work-Function Strategie wäre optimal, denn das Paging-Problem als Spezialfall des Server-Problems besitzt die untere Schranke k (siehe Satz 6.1). Zuerst führen wir die *Work-Function* formal ein.

Definition 7.4 Sei (X, d) ein metrischer Raum. Zu Anfang seien die k Server auf den Elementen $a_1^0, \ldots, a_k^0 \in X$ platziert, wobei $A_0 = \{a_1^0, \ldots, a_k^0\}$ eine Multimenge ist, es ist also erlaubt, dass mehr als ein Server auf einem Element von X stehen kann. Die Folge $(r_1, \ldots, r_t, \ldots)$ von Anforderungen liege vor.

(a) Für eine Multimenge $M \subseteq X$ mit k Elementen definieren wir die Work-Function w_t zum Zeitpunkt t durch

$$w_t(M) = \left\{ \begin{array}{l} \text{die minimale Strecke einer Bewegung} \\ \text{aller Server von } A_0 \text{ nach } M \text{ unter} \\ \text{Erfüllung der Anforderungen } r_1, \ldots, r_t \end{array} \right.$$

(b) Wir definieren die Work-Function Strategie iterativ und nehmen an, dass die Work-Function Strategie nach dem Abarbeiten der Anforderungen $r_1, r_2, \ldots, r_{t-1}$ die k Server zu den Standorten in der Menge A_{t-1} bewegt hat. Zur Erfüllung der Anforderung r_t wird ein Element $a \in A_{t-1}$ bestimmt, das

$$w_{t-1}\big(A_{t-1} \setminus \{a\} \cup \{r_t\}\big) + d(a, r_t)$$

minimiert und der Server auf a wird nach r_t bewegt.

Wir geben eine intuitive Erklärung. Der optimale off-line Algorithm Opt wird die Anforderungsfolge $r_1, \ldots, r_t, \ldots, r_n$ erfüllen, indem die Server von A_0 zu einer Menge A durch eine

Bewegung minimaler Länge überführt werden. Opt wird also die die Funktion $w_n(A)$ minimieren.

Der Work-Function Algorithmus orientiert sich also an dem optimalen Algorithmus Opt, kennt aber natürlich die Menge A der von Opt erreichten Zielpositionen nicht. Aber wir können tatsächlich annehmen, dass der Work-Function Algorithmus ebenfalls in der Menge A endet. Ist dies nicht der Fall, dann fügen wir Anforderungen an die vom Work-Function Algorithmus nicht erreichten Elemente aus A hinzu: Diese Anforderungen erhöhen die Kosten von Opt nicht.

Angenommen, der Work-Function Algorithmus hat die Konfiguration A_{t-1} erreicht und erhält eine neue Anforderung r_t. Dann wird ein Server auf Element a die Anforderung r_t erfüllen, d.h. zur Position r_t bewegt. Wir erhalten eine neue Konfiguration

$$A_t = A_{t-1} \setminus \{a\} \cup \{r_t\}.$$

Welchen Server sollen wir auswählen? Da wir unsere Strategie mit einer optimalen off-line Strategie vergleichen müssen, versuchen wir, eine Konfiguration A_t zu erreichen, die vom Standpunkt eines optimalen off-line-Algorithmus unter den möglichen Nachfolgekonfigurationen von A_{t-1} die geringsten Kosten hat. Allerdings darf der Übergang von A_{t-1} nach A_t für den on-line Algorithmus nicht zu teuer sein: Das Einfügen der on-line Kosten $d(x, r_t)$ sorgt für einen Kompromiss.

Beispiel 7.3 Wir wenden den Work-Function Algorithmus auf das Paging-Problem an. Da hier $d(u, v) = 1$ für alle verschiedenen Seiten u und v gilt, wird diejenige Seite aus A_{t-1} ausgelagert, die

$$w_{t-1}\big(A_{t-1} \setminus \{a_j\} \cup \{r_t\}\big)$$

unter allen Seiten $a_j \in A_{t-1}$ minimiert. Wenn $A_t = \{a_1, \ldots, a_k\}$ und die Anforderungen $r_1, r_2, \ldots, r_t$ zu erfüllen sind, dann ist $w_{t-1}(A_t)$ die minimale Anzahl von Strafpunkten, die eine off-line Strategie für die Erfüllung der Anforderungen

$$r_1, r_2, \ldots, r_t, a_1, \ldots, a_k$$

erhält, da diese Folge die Anforderungen $r_1, \ldots, r_t$ erfüllt und dabei die Server nach $a_1, \ldots, a_k$ bewegt.

Die LFD Strategie (longest forward distance) ist eine optimale off-line Strategie. LFD lagert stets die Seite aus, deren nächste Anforderung am weitesten in der Zukunft liegt.

Aufgabe 65

(a) Zeige, dass LFD eine optimale off-line-Strategie ist.

(b) Zeige, dass LFD bei einer beliebigen Folge von n Seiten aus einer Menge der Größe $k+1$ höchstens $\frac{n}{k}$ Seitenfehler macht.

Um

$$w_{t-1}\big(A_{t-1} \setminus \{a_i\} \cup \{r_t\}\big) \text{ und } w_{t-1}\big(A_{t-1} \setminus \{a_j\} \cup \{r_t\}\big)$$

zu vergleichen, genügt ein Vergleich nach Abarbeiten der Folgen

$$r_1, \ldots, r_{t-1}, a_1, \ldots, a_{i-1}, r_t, a_{i+1}, \ldots, a_k \text{ und } r_1, \ldots, r_{t-1}, a_1, \ldots, a_{j-1}, r_t, a_{j+1}, \ldots, a_k$$

durch LFD.

Behauptung 7.1 *Wenn die letzte Nachfrage nach a_i in $r_1, \ldots, r_t$ vor der letzten Nachfrage nach a_j liegt, dann gilt*

$$w_{t-1}\big(A_{t-1} \setminus \{a_i\} \cup \{r_t\}\big) \leq w_{t-1}\big(A_{t-1} \setminus \{a_j\} \cup \{r_t\}\big).$$

Beweis: Übungsaufgabe. □

Der Work-Function Algorithmus lagert also die Seite aus, deren letzte Anforderung in der Folge $r_1, \ldots, r_t$ am weitesten zurückliegt. Mit anderen Worten, der Work-Function Algorithmus stimmt für das Paging Problem mit der LRU Strategie überein.

Die Server Bewegungen der Work-Function Strategie, im Gegensatz zum Spezialfall des Paging Problem, können im allgemeinen Fall nur mit aufwändigen Berechnungen bestimmt werden. Hilfestellung liefert die folgende Beobachtung.

Lemma 7.5 *Für die Work-Function gilt*

$$w_t(M) = \min_{a \in M}\{w_{t-1}\big(M \setminus \{a\} \cup \{r_t\}\big) + d(a, r_t)\}.$$

Beweis: Die Teilmenge $M \subseteq X$ ist gegeben. Um $w_t(M)$ zu bestimmen, müssen wir alle Server von A_0 nach M bewegen, so dass die Anforderungen $r_1, \ldots, r_t$ zwischenzeitlich erfüllt werden: Die minimale Länge einer solchen Bewegung stimmt mit $w_t(M)$ überein.

In einer optimalen Bewegung von A_0 nach M betrachten wir die Vorgängerkonfiguration M^* von M, die die letzte Anforderung r_t erfüllt. Auf M^* folgen Bewegungen der Server in $M^* \setminus M$ zu den Elementen in $M \setminus M^*$. Diese Bewegungen können vor der Erfüllung der Anforderung r_t bereits ausgeführt werden. Davon ausgenommen ist der Server, der die Nachfrage r_t erfüllt: Dieser Server muss von r_t zu einem Element $a \in M \setminus M^*$ bewegt werden. Wir können also annehmen, dass M^* die Form

$$M^* = M \setminus \{a\} \cup \{r_t\}$$

hat. Natürlich sollte eine optimale Bewegung die Vorgängerkonfiguration M^* so wählen, dass die optimale Bewegung von A_0 nach M^* sowie die Bewegung von r_t nach a minimal ist. Diese letzte Überlegung ist aber äquivalent zur Behauptung. □

Angenommen, wir haben $w_{t-1}(B)$ für alle k-elementigen Teilmengen $M \subseteq X$ berechnet. Dann ist

$$w_t(M) = \begin{cases} w_{t-1}(M) & \text{falls } r_t \in M, \\ \min_{a \in M}\{w_{t-1}\big(M \setminus \{a\} \cup \{r_t\}\big) + d(a, r_t)\} & \text{sonst.} \end{cases}$$

Satz 7.6 *Der Work-Function Algorithmus hat einen Wettbewerbsfaktor von höchstens $2k-1$ und jeder on-line Algorithmus für das k-Server-Problem hat mindestens den Wettbewerbsfaktor k.*

Wir geben nur eine Skizze des Vorgehens und verweisen für das vollständige Argument auf [Ko]. Mit Lemma 7.5 folgt

$$w_t(A_{t-1}) = \min_{a \in A_{t-1}} \left\{w_{t-1}\big(A_{t-1} \setminus \{a\} \cup \{r_t\}\big) + d(a, r_t)\right\}. \tag{7.1}$$

Angenommen, das Minimum wird von $a = a_t$ angenommen. Beachte, dass $A_t = A_{t-1} \setminus \{a_t\} \cup \{r_t\}$ als Konsequenz von (7.1) gilt, und wir erhalten deshalb

$$w_t(A_{t-1}) = w_{t-1}(A_t) + d(a_t, r_t). \tag{7.2}$$

Um nach A_{t-1} unter Bedienung aller Anforderungen zu gelangen, ist es also optimal, die Server zuerst nach A_t unter Bedienung aller Anforderungen zu bewegen und dann den Server auf r_t nach a_t zu bewegen. Beachte, dass sich die Work-Funktion Strategie spiegelbildlich verhält, da sie von Konfiguration A_{t-1} nach Konfiguration A_t wandert und den Server auf a_t nach r_t bewegt.

Sei r_n die letzte Anforderung und bezeichne $A_0, A_1, \ldots, A_t, \ldots, A_n = A$ die Konfigurationen der Work-Function Strategie. Dann verursacht Opt die Kosten $w_n(A_n)$. Da $w_0(A_0) = 0$, folgt

$$w_n(A_n) = w_n(A_n) - w_0(A_0).$$

Desweiteren ist $w_n(A_n) - w_0(A_0) = \sum_{t=1}^{n} \big(w_t(A_t) - w_{t-1}(A_{t-1})\big)$ und deshalb ist

$$w_n(A_n) = \sum_{t=1}^{n} \big(w_t(A_t) - w_{t-1}(A_{t-1})\big). \tag{7.3}$$

Dieser Darstellung der Kosten des optimalen off-line Algorithmus stellen wir die Kosten der Work-Funktion Strategie gegenüber. Zuerst ist $w_{t-1}(A_t) = w_t(A_t)$, denn die Anforderung $r_t \in A_t$ ist kostenfrei. Aus (7.2) folgt somit

$$w_t(A_{t-1}) = w_t(A_t) + d(a_t, r_t)$$

und die Kosten der Work-Function Strategie werden durch

$$\sum_{t=1}^{n} d(a_t, r_t) = \sum_{t=1}^{n} w_t(A_{t-1}) - w_t(A_t) = w_0(A_0) - w_n(A_n) + \sum_{t=1}^{n} w_t(A_{t-1}) - w_{t-1}(A_{t-1})$$

beschrieben. Das Ziel der weiteren Analyse ist die Schranke

$$\sum_{t=1}^{n} \max_{M} \{w_t(M) - w_{t-1}(M)\} \leq 2k \cdot w_n(A_n). \tag{7.4}$$

Wir haben einen wesentlichen konzeptionellen Fortschritt erreicht, da die einzelnen Schritte $A_0, A_1, \ldots, A_t, \ldots$ der Work-Function Strategie nicht mehr nachvollzogen werden müssen. Vielmehr haben wir die Analyse der Work-Function Strategie durch die Analyse der Work-Function ersetzt.

Offenes Problem 3

Zeige, dass der Work-Function Algorithmus tatsächlich den Wettbewerbsfaktor k besitzt. Fortschritte werden in [Ko] berichtet.

Aufgabe 66

Wir betrachten den Liniengraph auf den Knoten $\{1, \ldots, k+1\}$ indem der Knoten i nur mit den Knoten $i+1$ und Knoten $i-1$ verbunden ist. Dies gilt mit Ausnahme des Knotens 1, der nur mit 2 verbunden ist, und mit Ausnahme des Knoten $k+1$, der nur mit k verbunden ist.

Der Wettbewerbsfaktor eines Algorithmus ist definiert wie vorher, wobei allerdings nur Anfragefolgen erlaubt sind, die Wegen im Liniengraphen entsprechen.

(a) Bestimme den Wettbewerbsfaktor von LRU und FIFO.

(b) Zeige, dass der Wettbewerbsfaktor von Arbitrary (lagere eine beliebige Seite aus) und Marking $\Omega(k)$ ist.

Aufgabe 67

Einem Paging Algorithmus mit Speicher k wird eine gleichverteilt zufällige Folge von Seiten aus $\{1, \ldots, k+1\}$ gegeben. Bestimme den erwarteten Wettbewerbsfaktor, d.h. den Quotienten aus erwarteten Kosten des Algorithmus und erwarteten optimalen Kosten für

(a) den Flush-when-Full Algorithmus,
(b) den Arbitrary "Algorithmus", bei dem bei einem Seitenfehler eine beliebige Seite verdrängt wird.

Aufgabe 68
Eine beliebige Verteilung D auf der Schlüsselmenge $\{1, \ldots, n\}$ sei gegeben. Zeige, dass Splay-Bäume „auf D" einen konstanten Wettbewerbsfaktor besitzen.

Aufgabe 69

(a) Beim *Traveling-Salesman-Problem mit Dreiecksungleichung* erfüllt die Distanzfunktion die Dreiecksungleichung, d.h. für alle Orte V_i, V_j, V_k gilt:

$$d(V_i, V_j) + d(V_j, V_k) \geq d(V_i, V_k).$$

Gib einen Polynomialzeit-Algorithmus an, der eine Tour berechnet, welche höchstens doppelt so lang ist wie das Optimum. Hinweis: Verwende einen minimalen Spannbaum.

(b) Beim on-line Traveling-Salesman-Problem ist ein metrischer Raum gegeben mit einem ausgezeichneten Punkt H. On-line-Eingaben sind Paare (V_i, t_i), wobei V_i ein Punkt im metrischen Raum ist und $t_i \in \mathbb{N}$ der Zeitpunkt, zu dem der Punkt V_i dem Handlungsreisenden bekanntgemacht wird. Der Reisende beginnt bei H, muss alle Punkte anfahren und dann wieder bei H ankommen. Dabei fährt er pro Zeiteinheit eine Entfernungseinheit. Gültige Touren dürfen eine Eingabe (V_i, t_i) erst nach t_i erfüllen und das gilt auch für off-line-Algorithmen. Beachte, dass mehrere Eingaben zu einer Zeit ankommen dürfen und dass nicht bekannt ist, wieviele Eingaben insgesamt kommen. Kosten entsprechen der Zeit, um eine Anfragefolge zu erfüllen. Gib einen on-line Algorithmus an, der einen konstanten Wettbewerbsfaktor hat. Jede Ausgabe des Algorithmus' soll effizient berechenbar sein.

Aufgabe 70
Ein Graph wird on-line Kante für Kante bekannt gegeben. Es ist das Ziel eines on-line Algorithmus, die Kanten zu färben. Dabei dürfen zwei Kanten, welche zum selben Knoten inzident sind, nicht dieselbe Farbe bekommen. Als Kosten wird die Anzahl benutzter Farben verwandt.

(a) Gib einen on-line Algorithmus an, der mit $2\Delta - 1$ Farben auskommt, wenn der Graph einen maximalen Grad von Δ besitzt.

(b) Zeige, dass jeder on-line Algorithmus für das Problem im worst case mindestens $2\Delta - 1$ Farben benötigt.

HINWEIS: Ein Gegner erzeugt zunächst einen Graphen aus Sternen (in einem Stern ist ein zentraler Knoten mit $\Delta - 1$ anderen Knoten verbunden, welche nicht untereinander verbunden sind). Er erzeugt solche Sterne, bis Δ viele Sterne die gleiche Farbenmenge in der vom on-line Algorithmus gegebenen Färbung aufweisen (wieso ist das möglich?). Dann kann der Gegner den Algorithmus zu vielen Farben zwingen.

Aufgabe 71
Betrachte das Umzugsproblem: gegeben ist die Fläche $[0, 1] \times [0, \infty)$, in welche Rechtecke mit Seitenlängen kleiner als 1 eingepackt werden sollen. Dabei kommen die Rechtecke online von „oben" (der offenen Seite) und müssen an ihren Platz bewegt werden (eventuell an anderen Rechtecken vorbei). Man denke an einen Lastwagen, der mit Paketen beladen werden soll. Dabei müssen Rechtecke parallel zu den Seiten angeordnet werden, dürfen aber gedreht werden. Sind sie platziert, darf ihre Position nicht mehr verändert werden. Es soll minimale Höhe verbraucht werden.

Zeige, dass es einen On-Line-Algorithmus mit konstantem Wettbewerbsfaktor gibt. Dabei ist beim Wettbewerbsfaktor eine additive Konstante erlaubt. Das bedeutet, dass der On-Line-Algorithmus eine Höhe von $k \cdot \text{opt} + O(1)$ für eine Konstante k bei optimaler Höhe opt verwenden darf.

Hinweis: Rotiere Rechtecke immer auf die schmalere Seite. Reserviere einen eigenen horizontalen Streifen der Fläche für jedes breite Rechteck (z.B. solche $\geq \frac{1}{4}$). Schmalere Rechtecke, welche sich in der Höhe nicht zu sehr unterscheiden, gruppiere ebenfalls in horizontalen Streifen. Besonders breite Rechtecke ($\geq \frac{3}{4}$) blockieren den Weg nach unten, verrechne Lücken unter ihnen mit ihrer Fläche.

Kapitel 8

Auswahl von Experten

Im maschinellen Lernen möchte man Prognosen für die Zukunft erstellen und Lernalgorithmen lassen sich damit automatisch als on-line Algorithmen auffassen. Wir stellen zwei wichtige Algorithmen, die Weighted Majority und die Winnow Strategie vor, für die relativ kleine Wettbewerbsfaktoren bestimmt werden können.

8.1 Der Weighted Majority Algorithmus

Wir müssen eine Folge von Ja/Nein Entscheidungen treffen. Zu jedem Zeitpunkt steht uns eine Menge E von Experten zur Verfügung, wobei jeder Experte zu jedem Zeitpunkt entweder die Entscheidung „Ja“ oder die Entscheidung „Nein“ empfiehlt. Nachdem wir, basierend auf allen Empfehlungen, eine Entscheidung getroffen haben, wird uns die richtige Entscheidung mitgeteilt.

> Können wir, ohne den die Entscheidungen betreffenden Sachverhalt überhaupt zu kennen, eine Strategie entwickeln, deren Entscheidungen fast so gut sind wie die Empfehlungen des bisher besten Experten?

Das Problem besteht also darin, innerhalb kürzester Zeit die Qualität des bisher besten Experten zu erreichen. Beachte, dass sich bisher optimale Experten in der Zukunft signifikant verschlechtern können und wir werden in jeder Runde[1] neu überdenken müssen, welcher Empfehlung wir folgen.

In unserer ersten Strategie nehmen wir an, dass n Experten Empfehlungen abgeben. Unsere Strategie wird dem $i$ten Experten ein Gewicht w_i zuweisen, dass die Güte seiner Empfehlungen wiederspiegelt.

Algorithmus 8.1 Eine einfache Version der **Weighted Majority Algorithmus**

(1) Setze $w_i = 1$ für alle Experten i.

(2) Wenn Experte i die Empfehlung $x_i \in \{\text{Ja}, \text{Nein}\}$ abgibt, dann treffe die Entscheidung „Ja“, falls

$$\sum_{i, x_i = \text{Ja}} w_i \geq \sum_{i, x_i = \text{Nein}} w_i$$

und ansonsten treffe die Entscheidung „Nein“.

[1] Eine Runde besteht aus den Empfehlungen der Experten, unserer Entscheidung und schließlich der Bekanntmachung der richtigen Entscheidung.

(3) Nach Erhalt der richtigen Entscheidung: Wenn Experte i eine falsche Emphehlung abgegeben hat, dann bestrafe ihn mit der Setzung $w_i = w_i/2$. Ansonsten lasse das Gewicht w_i unverändert.

Wie gut ist der Weighted Majority Ansatz? Sei W_t das Gesamtgewicht aller Experten vor Beginn von Runde t. Dann ist anfänglich $W_1 = n$. Wenn die in Runde t abgegebene Entscheidung falsch ist, dann haben sich Experten mit einem Gesamtgewicht von mindestens $W_t/2$ geirrt. Folglich ist

$$W_{t+1} \leq \frac{1}{2} \cdot \frac{W_t}{2} + \frac{W_t}{2} = \frac{3}{4} W_t.$$

Wenn wir f bis zum Zeitpunkt t falsche Entscheidungen getroffen haben, dann folgt insbesondere

$$W_t \leq n \cdot (\frac{3}{4})^f. \tag{8.1}$$

Können wir sehr viel schlechter sein als der beste Experte? Wenn der beste Experte bis zum Zeitpunkt t genau f_{opt} Fehler gemacht hat, dann ist sein Gewicht $2^{-f_{\text{opt}}}$. Insbesondere ist damit auch $2^{-f_{\text{opt}}} \leq W_t$. Wir kombinieren diese untere Schranke für W_t mit der oberen Schranke (8.1) und erhalten die Bedingung

$$(\frac{1}{2})^{f_{\text{opt}}} \leq n \cdot (\frac{3}{4})^f, \quad \text{bzw } (\frac{4}{3})^f \leq 2^{f_{\text{opt}}} \cdot n.$$

Wir logarithmieren und erhalten die Ungleichung

$$f \cdot \log_2(4/3) \leq f_{\text{opt}} + \log_2 n.$$

Wir können zusammenfassen:

Satz 8.2 *Es möge n Experten geben. Wenn der Weighted Majority Algorithmus bisher f falsche und der beste Experte f_{opt} falsche Entscheidungen getroffen haben, dann gilt*

$$f \leq \frac{1}{\log_2(4/3)} \cdot (f_{\text{opt}} + \log_2 n).$$

Algorithmus 8.1 erreicht damit, bis auf den additiven logarithmischen Term, den Wettbewerbsfaktor $\frac{1}{\log_2(4/3)}$.

Wir sind also nahe am optimalen Experten „dran“: Im Wesentlich brauchen wir $\log_2 n$ Zeit, um die Gewichte der Qualität der Experten entsprechend zu setzen, verfehlen aber leider nach dieser „Aufwärmzeit“ die Qualität des besten Experten um den Faktor $\frac{1}{\log_2(4/3)} \approx 2.41$. Wir können aber unsere Voraussagekraft verbessern, wenn wir die Aufwärmzeit verlängern.

Bevor wir diese Verbesserung beschreiben, betrachten wir aber eine weitere, sehr einfache Strategie: Warum zum Zeitpunkt t nicht einfach die Entscheidung des Experten übernehmen, der bisher die wenigsten Fehler gemacht hat? Weil die Vergangenheit die Zukunft nicht voraussagen kann: Ein bösartiger Gegner, die Zukunft, klassifiziert die Entscheidung des bisher besten Experten möglicherweise als falsch.

Aufgabe 72

Wir führen die Strategie des Wahl des besten Experten für t Schritte aus. Konstruiere eine Menge von n Experten, so dass sich die Strategie immer irrt, obwohl der beste Experte höchstens t/n Fehler macht.

Trotzdem ist die Grundidee, also die Auswahl des besten Experten, nicht schlecht. Tatsächlich können wir die einfache Version der Weighted Majority Version verbessern, wenn wir Experten zufällig, aber proportional zu ihrem gegenwärtigen Gewicht auswählen! Dieser Ansatz erlaubt sogar die Lösung eines allgemeineren Problems:

- Statt einer Ja/Nein Entscheidung erwarten wir diesmal allgemeinere Empfehlungen. Die Empfehlung eines Experten i zum Zeitpunkt t bewerten wir mit einer „Note“ c_i^t auf einer Skala von 0 (sehr gut) bis 1 (sehr schlecht).
- Unser Ziel ist demgemäß die Bestimmung eines Experten mit einer möglichst gut bewerteten Empfehlung.

Wir werden es natürlich nicht schaffen, in jeder Runde einen optimalen Experten zu ermitteln, sondern unser Ziel sollte sein, die Qualität eines bisher besten Experten zumindest approximativ zu erreichen, wenn wir *alle* bisherigen Empfehlungen in Betracht ziehen.

Algorithmus 8.3 Eine **randomisierte** Version von **Weighted Majority**

(1) Setze $w_i = 1$ für alle Experten i. Die Konstante ε wird aus dem Intervall $]0, 1/2[$ gewählt.

(2) Wähle einen Experten zufällig, wobei Experte i die Wahrscheinlichkeit

$$p_i = \frac{w_i}{\sum_{k=1}^{n} w_k}$$

erhält und übernimm seine Entscheidung.

(3) Berechne neue Gewichte: Setze $w_i = w_i(1 - \varepsilon \cdot c_i^t)$.
Kommentar: Beachte $\varepsilon \cdot c_i^t \in [0, 1/2[$.

Obwohl wir mehr verlangen, hat unsere Voraussagekraft zugenommen:

Satz 8.4 *Es möge n Experten geben. Wenn der randomisierte Weighted Majority Algorithmus bisher Entscheidungen mit den erwarteten Gesamtkosten K und der beste Experte Entscheidungen mit den minimalen Gesamtkosten K_{opt} getroffen hat, dann ist*

$$K \leq (1 + \varepsilon) \cdot K_{\text{opt}} + \frac{\ln n}{\varepsilon}.$$

Algorithmus 8.3 erreicht damit, bis auf den additiven logarithmischen Term, den Wettbewerbsfaktor $(1 + \varepsilon)$.

Beweis: w_i^t sei das Gewicht von Experte i zu Beginn von Runde t und $W_t = \sum_{i=1}^{n} w_i^t$ sei die Gewichtssumme aller n Experten zu Beginn von Runde t. Schließlich sind

$$K_t = \sum_{i=1}^{n} \frac{w_i^t}{W_t} \cdot c_i^t$$

die erwarten Kosten unserer Entscheidung zum Zeitpunkt t. Wir beobachten zuerst, dass

$$\begin{aligned} W_{t+1} &= \sum_{i=1}^{n} w_i^t \cdot (1 - \varepsilon \cdot c_i^t) = \sum_{i=1}^{n} w_i^t - \varepsilon \cdot \sum_{i=1}^{n} w_i^t \cdot c_i^t \\ &= W_t - \varepsilon \cdot K_t \cdot W_t = W_t \cdot (1 - \varepsilon \cdot K_t) \end{aligned} \tag{8.2}$$

gilt. Um einen Zusammenhang herzustellen zwischen unseren erwarteten Gesamtkosten bis zum Zeitpunkt T und dem Gewicht W_{T+1} zum Zeitpunkt $T+1$, expandieren wir die Darstellung (8.2). Wir erhalten

$$W_{T+1} = W_1 \cdot \Pi_{t \leq T}(1 - \varepsilon \cdot K_t).$$

Wir lösen nach unseren erwarteten Kosten K_t auf, indem wir logarithmieren und beachten, dass $\log(1-x) \leq -x$ gilt:

$$\begin{aligned} \ln W_{T+1} &= \ln W_1 + \sum_{t \leq T} \ln(1 - \varepsilon \cdot K_t) \leq \ln n - \sum_{t \leq T} \varepsilon K_t \\ &= \ln n - \varepsilon \cdot K, \end{aligned} \tag{8.3}$$

wobei K unsere erwarteten Gesamtkosten sind. Nachdem wir gezeigt haben, dass hohe erwartete Kosten unsererseits das Gesamtgewicht nach unten drücken, zeigen wir jetzt, wie im Beweis von Satz 8.2, dass ein guter Experte das Gesamtgewicht nach oben zieht. Für jeden Experten i gilt

$$W_{T+1} \geq w_i^{T+1} = \Pi_{t \leq T}(1 - \varepsilon \cdot c_i^t)$$

Wir logarithmieren wieder, um an die Gesamtkosten $K(i)$ des $i$ten Experten heranzukommen. Diesmal benutzen wir, dass $\ln(1-x) \geq -x - x^2$ für $x \in [0, 1/2]$ gilt:

$$\begin{aligned} \ln W_{T+1} &= \sum_{t \leq T} \ln(1 - \varepsilon \cdot c_i^t) \geq -\sum_{t \leq T} (\varepsilon \cdot c_i^T + (\varepsilon \cdot c_i^T)^2) \\ &\geq -\sum_{t \leq T} (\varepsilon \cdot c_i^T + \varepsilon^2 \cdot c_i^T) = -(\varepsilon + \varepsilon^2) \cdot K(i). \end{aligned} \tag{8.4}$$

Wir haben hier benutzt, dass c_i^t im Intervall $[0, 1]$ liegt, um die Ungleichung $c_i^t \geq (c_i^t)^2$ anwenden zu können. Wir kombinieren (8.3) und (8.4), in dem wir den Experten i mit minimalen Gesamtkosten $K(i) = K_{\text{opt}}$ betrachten. Es ist

$$-(\varepsilon + \varepsilon^2) \cdot K_{\text{opt}} \leq \ln W_{T+1} \leq \ln n - \varepsilon \cdot K.$$

Wir erhalten die Behauptung nach Division durch $-\varepsilon$. □

In welchen Situationen können wir den randomisierten Weighted Majority Algorithmus anwenden? Angenommen, wir müssen ein sehr komplexes Optimierungsproblem lösen und haben verschiedene Heuristiken „gebaut“, die Lösungen unterschiedlicher Qualität erreichen. Wir nehmen an, dass wir die Qualität der Lösungen unmittelbar nicht miteinander vergleichen können: Wenn wir zum Beispiel Aktiengeschäfte tätigen wollen, dann wird erst die Zukunft zeigen, wie gut welche Entscheidungen waren.

Wir interpretieren die Heuristiken als Experten und bewerten die jeweils berechneten Lösungen auf der $[0, 1]$-Skala. Wir wenden dann den Weighted Majority Algorithmus an und bestimmen eine Gewichtung der Heuristiken, die sich dann fast-optimal auf die Daten einstellt.

8.2 On-line Auswahl von Portfolios

Die kontinuierliche Vermögensanlage ist ein Musterbeispiel eines on-line Problems. Lässt sich der Weighted Majority Ansatz nutzen, um sinnvolle Anlagemethoden zu entwerfen? Sicherlich können wir die verschiedenen Anlagemethoden als Experten modellieren, allerdings ist die

Auswahl einer Anlagemethode, ob durch Mehrheitsentscheidung oder durch zufällige Wahl, höchst riskant, wenn das Risiko eines Totalverlusts nicht ausgeschlossen werden kann.

Stattdessen gehen wir einen anderen Weg: Um das Risiko zu begrenzen, verteilen wir das zu investierende Vermögen V gleichmässig über eine große Zahl N von verschiedensten Portfolios und managen ein Portfolio nach den Regeln des jeweilig zugrunde liegenden Investmentansatzes. Wir modellieren dann die Portfolios durch Experten und weisen dem $i$ten Portfolio anfänglich das Gewicht $w_i^1 = V/N$ zu; damit geben wir wieder, dass alle N Portfolios mit dem gleichen Vermögen V/N starten. Wie sollten wir die Gewichte neu setzen, wenn die Portfolios in regelmässigen Abständen neu justiert werden? Darüber brauchen wir uns keine Gedanken machen, da der Markt dies für uns tut: Setze w_i^t gleich dem Vermögen des $i$ten Portfolios vor der t.ten Neujustierung.

Wir konzentrieren uns hier auf den CRP-Ansatz (Constant Rebalanced Portfolio). Wenn A eine Menge von Aktien und $p = (p_a \mid a \in A)$ eine Verteilung auf den Aktien in A ist, behält der CRP-Ansatz dieselbe Verteilung p des gegenwärtigen Vermögens über alle Anlageperioden bei.

Beispiel 8.1 Wir nehmen zwei Aktien ($A = \{1, 2\}$) an, wobei sich die erste Aktie nicht bewegt, während sich die zweite Aktie alternierend erst halbiert und dann verdoppelt. Offensichtlich bringt es nichts, das verfügbare Vermögen nur in einer Aktie anzulegen, da nach je zwei Anlageterminen das alte Vermögen wieder erreicht wurde. Stattdessen verteilen wir das Vermögen gleichgewichtet auf die beiden Aktien, wählen also die Verteilung $p = (0.5, 0.5)$.

Wenn V das gegenwärtige Vermögen ist, dann ist V vor dem zweiten Anlagetermin auf $V \cdot (1/2 + (1/2) \cdot (1/2)) = 3 \cdot V/4$ gesunken, um vor dem dritten Anlagetermin wieder auf $V \cdot (3/8 + 2 \cdot 3/8) = 9 \cdot V/8$ zu steigen. Vor dem Anlagetermin $2n + 1$ ist das Vermögen damit exponentiell auf $(\frac{9}{8})^n \cdot V$ angewachsen!

Wenn wir den CRP-Ansatz über längere Zeit verfolgen, dann führen verschiedene Verteilungen p also zu radikal verschiedenen Vermögensentwicklungen. Wir setzen uns das Ziel, die *erwartete Vermögensentwicklung* des CRP-Ansatzes zu erreichen.

Wenn wir die Experten-Analogie anwenden, dann sollten wir idealerweise mit unendlichen vielen Experten arbeiten, nämlich einem Experten für jede Verteilung. Natürlich können wir uns unendlich viele Experten nicht leisten, sondern wählen stattdessen eine genügend große Anzahl N von Verteilungen zufällig aus. Der Universal Portfolio Algorithmus von Cover [C] verfolgt im Wesentlichen diesen Ansatz, legt in jedem ausgewählten Portfolio dasselbe Vermögen an und rebalanciert jedes Portfolio für jeden Anlagetermin nach dem CRP-Ansatz, wobei aber keine Gelder zwischen verschiedenen Portfolios transferiert werden dürfen.

Wie gut ist der Universal Portfolio Algorithmus? Wir nehmen an, dass wir mit m Aktien über einen Zeitraum von n Anlageperioden arbeiten. Wenn opt_n das optimale Vermögen und e_n das erwartete Vermögen nach n Anlageperioden ist, dann gilt:

Fakt 8.1 *[C]*

$$e_n \geq \frac{\mathrm{opt}_n}{(n+1)^{m-1}}.$$

Damit kann das erwartete Verhalten potentiell sehr viel schlechter als das optimale Verhalten sein, aber überraschenderweise ist der maximale Verlustfaktor höchstens polynomiell und nicht exponentiell in der Anzahl der Anlageperioden. Der durchschnittliche maximale relative Verlustfaktor pro Anlageperiode ist höchstens

$$\left((n+1)^{m-1}\right)^{1/n} \approx \left(n^{m-1}\right)^{1/n} = n^{(m-1)/n} = 2^{\frac{(m-1)\cdot \log_2 n}{n}}$$

und damit für eine genügend große Zahl n von Anlageperioden nur unwesentlich größer als Eins: Legt die optimale CRP-Strategie über einen genügend langen Zeitraum um den Faktor a_n^n für $a_n > 1$ zu, dann wird der Universal Portfolio Ansatz einen Wertzuwachs um den Faktor b_n^n erreichen, wobei $1 < b_n < a_n$ und $\lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n$ gilt. Der Universal Portfolio Ansatz schneidet somit auf den zweiten Blick nicht schlecht ab:

> Die erwartete CRP-Vermögensentwicklung kann bis auf polynomielle Faktoren mit der besten CRP-Vermögensentwicklung mithalten: Damit gewährleistet die Neugewichtung, dass genügend viele zufällig gewählte Verteilungen einen nicht zu großen Abstand von der optimalen Verteilung haben.

8.3 Der Winnow Algorithmus

Nehmen wir wieder an, dass n Experten zur Verfügung stehen. Da ein Experte sich für einige Entscheidungen möglicherweise nicht kompetent fühlt, geben wir diesmal sogar die Möglichkeit der Enthaltung, statt Experten arbeiten wir also wirklich mit Spezialisten. Auch unsere Erwartungshaltung wächst: Statt ungefähr so gut zu sein wie ein bester Spezialist, möchten wir die Voraussageleistung einer besten **Auswahl** E von Spezialisten annähernd erreichen. Wie ist das Votum einer Auswahl, bzw. Menge von Spezialisten zu interpretieren? Wenn alle sich nicht enthaltenden Spezialisten in E einstimmig sind, dann wird dieses einstimmige Votum übernommen, in allen anderen Fällen oder wenn das einstimmige Votum falsch ist, wird die Menge E mit einem Strafpunkt belegt.

Um mit allen Mengen E mithalten zu können, bauen wir für jede Menge E einen „Superexperten“ i_E, der ein einstimmiges Votum „seiner“ Spezialisten übernimmt und sich sonst enthält. Wir wenden Algorithmus 8.1 auf die 2^n Superexperten an und können somit mit der Voraussagekraft jeder Menge E mithalten. (Dabei setzen wir voraus, dass jede Enthaltung oder falsche Empfehlung von i_E mit einem Strafpunkt geahndet wird.) Was ist das Problem? Pro Runde benötigt Algorithmus 8.1 die Laufzeit mindestens $\Omega(2^n)$ und ist damit nicht mehr praktikabel. Haben wir trotzdem noch eine Chance?

Wir stellen als Nächstes den Winnow[2] Algorithmus vor, der auch mit dieser schwierigeren Fragestellung klarkommt. Wir müssen allerdings die Voraussageleistung von E ein wenig strenger bewerten.

Definition 8.5 Sei E eine Menge von Spezialisten. Wir vegeben in einer Runde

- k Strafpunkte, wenn sich genau k Spezialisten aus E irren,
- bzw. einen Strafpunkt, wenn sich alle Spezialisten aus E enthalten.

Wenn s_t die Anzahl der Strafpunkte von E in Runde t ist, dann ist

$$\text{Strafe}_T(E) = \sum_{t=1}^{T} s_t$$

die Gesamtstrafe für E nach T Runden.

Algorithmus 8.6 **Der Winnow-Algorithmus**

[2] To winnow: den Spreu vom Weizen trennen.

(1) Setze $w_1 = \cdots = w_n = 1$.

(2) Wenn eine Entscheidung zu treffen ist, dann gibt jeder Spezialist i bekannt, ob er sich enthält ($x_i = 0$) oder nicht ($x_i = 1$).

(2a) Winnow enthält sich, falls

$$\sum_{i=1}^{n} w_i \cdot x_i < n$$

gilt. Nachfolgend werden die Gewichte aller Spezialisten, die sich *nicht* enthalten haben, verdoppelt.

(2b) Sonst ist $\sum_{i=1}^{n} w_i \cdot x_i \geq n$ und Winnow trifft eine Mehrheitsentscheidung: Wenn $y_i \in \{\text{Ja}, \text{Nein}\}$ die Empfehlung von Spezialist i ist, dann entscheidet Winnow auf „Ja“, wenn

$$\sum_{i, x_i=1, y_i=\text{Ja}} w_i \cdot x_i \geq \sum_{i, x_i=1, y_i=\text{Nein}} w_i \cdot x_i$$

und ansonsten auf „Nein“. Nachfolgend werden die Gewichte aller Spezialisten, die die falsche Entscheidung unterstützt haben, halbiert.

Wir werden jetzt sehen, dass Winnow dann eine gute Leistung abliefert, wenn es kleine, aber gute Mengen von Spezialisten gibt.

Satz 8.7 *Es möge insgesamt n Spezialisten geben. Sei E eine beliebige Menge von Spezialisten. Dann macht Winnow nach T Runden höchstens*

$$5 \cdot \text{Strafe}_T(E) + 5 \cdot |E| \cdot \lceil \log_2 n \rceil + 4$$

Fehler, wobei ein Fehler entweder eine Enthaltung oder eine falsche Entscheidung ist.

Beweis: Wir führen Winnow für insgesamt T Runden aus und vergleichen die Voraussagen von Winnow und der Spezialistenmenge E. Für jeden Spezialisten $i \in E$ sei f_i die Anzahl der bisherigen falschen Voraussagen von i. Schließlich möge es genau e Zeitpunkte geben, an denen sich alle Spezialisten aus E enthalten.
Schritt 1: Wir beschränken zuerst die Anzahl e^* der Enthaltungen von Winnow, indem wir einen Zusammenhang mit den Fehlern und Enthaltungen von E herstellen. Wir beachten, dass Winnow sich nicht enthält, wenn mindestens ein Spezialist i ein großes Gewicht besitzt, d.h. wenn $w_i \geq n$ gilt, und wenn sich dieser Experte nicht enthält. Das Gewicht w_i wird nach einer Fehlentscheidung von Spezialist i halbiert und nach einer Enthaltung von Winnow verdoppelt, solange sich i nicht selbst enthalten hat. Wie entwickeln sich die Gewichte für die Spezialisten in E?

Angenommen ein Spezialist $i \in E$ hat sich mindestens $f_i + \log_2 n$ *nicht* enthalten, während sich Winnow jedesmal enthalten hat. Dann ist sein Gewicht w_i, unter Einrechnung der Halbierungen durch die f_i falschen Entscheidungen, auf mindestens n angestiegen und dieser Spezialist allein verhindert eine Enthaltung, wann immer er sich nicht enthält.

In jeder von $e^* - e$ Enthaltungen von Winnow wird sich aber mindestens ein Spezialist aus E “bekennen“, sich also nicht enthalten. Jeder Spezialist $i \in E$ erreicht aber nach $f_i + \log_2 n$

„Bekennungsschritten“ das kritische Gewicht n und verhindert danach Enthaltungen von Winnow. Also haben wir

$$e^* \leq e + \sum_{i \in E} f_i + |E| \cdot \lceil \log_2 n \rceil \tag{8.5}$$

$$= \text{Strafe}_T(E) + |E| \cdot \lceil \log_2 n \rceil \tag{8.6}$$

nachgewiesen.

Schritt 2: Um die Anzahl der Fehlentscheidungen von Winnow zu beschränken, stellen wir einen Zusammenhang zu den Enthaltungen von Winnow her. Sei W_t die Summe aller Gewichte zu Beginn von Runde t. Wir machen die folgenden Beobachtungen:

(1) Es ist $W_1 = n$ und $W_t > 0$ für alle Zeitpunkte t.

[2) Wenn Winnow sich in Runde t enthält, dann werden nur Gewichte der sich nicht enthaltenden Spezialisten verdoppelt. Es gilt aber $\sum_{i=1}^{n} w_i x_i < n$, und deshalb folgt $W_{t+1} \leq W_t + n$.

(3) Wenn Winnow in Runde t eine Fehlentscheidung trifft, dann ist andererseits $\sum_{i=1}^{n} w_i x_i \geq n$, und das Gesamtgewicht der sich irrenden Spezialisten ist mindestens $n/2$. Das Gesamtgewicht der sich irrenden Spezialisten wird halbiert, und deshalb folgt $W_{t+1} \leq W_t - \frac{n}{4}$.

Wenn f^* die Anzahl der Fehlentscheidungen von Winnow ist, dann muss also $f^* \leq 4e^* + 4$ gelten, denn ansonsten wäre das Gesamtgewicht negativ. (Da das Gesamtgewicht zu Anfang n ist, treiben 4 Fehlentscheidungen das Gewicht auf Null.) Also ist die Anzahl der Enthaltungen und Fehlentscheidungen durch

$$e^* + f^* \leq 5e^* + 4 \leq 5 \cdot \text{Strafe}_T(E) + 5 \cdot |E| \cdot \lceil \log_2 n \rceil + 4$$

beschränkt. □

Unsere Bestrafung der Spezialistenmenge E ist unfair, da wir eine Fehlentscheidung mit der Anzahl der sich irrenden Spezialisten in E bestrafen. Eine faire Bewertung bestraft eine Fehlentscheidung nur einmal. Wenn f also die Anzahl der Fehler von E ist, dann folgt $\text{Strafe}_T(E) \leq e + f \cdot |E| \leq (e + f) \cdot |E|$ und die Anzahl der Enthaltungen und Fehlentscheidungen von Winnow ist durch

$$O((e + f + \log_2 n) \cdot |E|)$$

beschränkt:

Korollar 8.8 *Winnow besitzt den Wettbewerbsfaktor $O(|E|)$.*

Natürlich hätten wir auch die Mehrheitsentscheidung in Schritt (2b) durch die zufällige Wahl eines sich nicht enthaltenden Spezialisten ersetzen können.

Aufgabe 73
Entwerfe einen Lernalgorithmus, der eine unbekannte Disjunktion $D = x_{i_1} \vee \cdots \vee x_{i_r}$ nach möglichst wenigen Gegenbeispielen erlernt.

Hinweis: Interpretiere jedes Literal x_j als einen Experten. Das Ziel ist dann die exakte Bestimmung der D zugrundeliegenden Expertenmenge.

Aufgabe 74
Statt einer Disjunktion ist diesmal eine unbekannte Threshold-Funktion $\sum_{j=1}^{r} x_{i_j} \geq t$ zu erlernen.

Kapitel 9

Selbst-organisierende Datenstrukturen

9.1 Amortisierte Laufzeit

Wir führen den Begriff der amortisierten Laufzeit anhand zweier einfacher Beispiele ein.

Beispiel 9.1 **Binäre Zähler** Wir möchten eine Folge von Inkrement-Operationen auf einen anfänglich auf 0 gesetzten binären Zähler ausführen. Insbesondere, wenn i der aktuelle Zählerstand ist und die Operationen Inkrement(i) auszuführen ist, dann benötigen wir sämtliche Bitoperationen, die die Binärdarstellung von i in die Binärdarstellung von $i+1$ umwandeln. Diese Anzahl benötigter Bitoperationen ist offenbar genau k, wobei $k-1$ die maximale Anzahl von aufeinanderfolgenden Einsen beginnend mit dem niedrigstwertigen Bit ist:

$$1001\ \underbrace{0111}_{k \text{ Bits}} \ \rightarrow\ 1001\ \underbrace{1111}_{k \text{ Bits}}$$

Viele Inkrement-Operationen sind sehr „billig“: Wenn der Zählerinhalt z gerade ist, genügt das Flippen des niedrigstwertigen Bits. Es gibt aber auch sehr „teure“ Inkrement-Operationen: Wenn $z \equiv 2^{k-1} - 1 \pmod{2^k}$, müssen wir k Bits flippen.

Wir beginnen mit Zählerinhalt 0 und führen i Inkrement-Operationen durch. Wie oft werden Bits geflippt? Das niedrigstwertige Bit 0 wird bei jeder Inkrement-Operation geflippt, Bit 1 wird bei jeder zweiten Inkrement-Operation geflippt und allgemein wird Bit k jedes 2^k-te Mal geflippt. Insgesamt werden also höchstens

$$\sum_{k=0}^{\lfloor \log_2 i \rfloor} \frac{i}{2^k} \leq \sum_{k=0}^{\infty} \frac{i}{2^k} = 2i$$

Bits geflippt.

Im folgenden zweiten Beweis beschreiben wir das *Buchhalter*-Argument. Hier ist die grundlegende Idee, die Laufzeit auf die Operationen umzulegen. Dabei sind Operationen nicht nur für die von ihnen direkt verursachten Schritte zu belasten, sondern auch für die indirekt verursachten, später auszuführenden Schritte. In unserem Beispiel belasten wir eine Inkrement-Operation mit zwei Kosteneinheiten: Jede Inkrement-Operation lassen wir eine Einheit für die eine, neu eingeführte 1 bezahlen. Die verbleibende Einheit ist in der Zukunft zu bezahlen, wenn nämlich die gerade eingeführte 1 später in eine 0 zu flippen ist.

Betrachten wir jetzt die Durchführung eines beliebigen Inkrement-Schrittes. Das Flippen einer jeden 1 in eine 0 ist bereits durch die Operation bezahlt worden, die diese 1 eingeführt hat. Die aktuelle Operation muss also nur für die Neueinführung ihrer 1 sowie für die spätere Umkehrung in eine 0 bezahlen: Der Etat von 2 Kosteneinheiten ist somit ausreichend und i Inkrement-Operationen laufen in Zeit höchstens $2 \cdot i$.

Wir geben einen dritten Beweis, um das Konzept der *Potentialfunktion* einzuführen. Dazu beachten wir, dass billige Operationen verantwortlich für teure Operationen sind, denn eine teure Operation muss die von billigen Operationen eingeführten Einsen flippen. Diese billigen Operationen erhöhen die Gefahr (bzw. das Potential) für eine nachfolgende, teure Operation. Um das aufgebaute Potential Φ vor Schritt k zu quantifizieren, setzen wir

$$\Phi(k) := \left[\begin{array}{l} \text{Anzahl der Einsen in der} \\ \text{Binärdarstellung von } k \end{array} \right].$$

Insbesondere ist $\Phi(0) = 0$ und $\Phi(k) \geq 0$. Eine Operation, die $r \geq 1$ Einsen flippt, senkt das Potential um $r - 1$, da eine 1 neu eingeführt wird und r Einsen verschwinden. Eine billige Operation, die keine Einsen flippt, erhöht das Potential um 1. Wir lassen die billigen (und damit potentialerhöhenden) Operationen für eine teure Operation bezahlen und definieren dazu die **amortisierten Kosten** der k-ten Operation ($k \geq 1$) durch

$$\text{Amortisierte-Kosten}_k := \text{Wirkliche-Kosten}_k \quad + \quad \underbrace{\Phi(k) - \Phi(k-1)}_{\text{Änderung des Potentials}},$$

wobei $\text{Wirkliche-Kosten}_k$ die wirklichen Kosten (Anzahl der zu flippenden Bits) der k-ten Operation sind. Die k-te Operation flippt $\text{Wirkliche-Kosten}_k - 1$ Einsen sowie eine Null und senkt das Potential um $\text{Wirkliche-Kosten}_k - 2$. Ihre amortisierten Kosten sind also

$$\text{Amortisierte-Kosten}_k = \text{Wirkliche-Kosten}_k - (\text{Wirkliche-Kosten}_k - 2) = 2.$$

Es gilt in diesem Fall $\text{Amortisierte-Kosten}_k = 2$ für alle k. Was nutzen uns diese Überlegungen? Wir können von den amortisierten auf die wirklichen Kosten zurückschließen, denn wir zeigen als Nächstes, dass

$$\sum_{k=1}^{i} \text{Wirkliche-Kosten}_k \leq \sum_{k=1}^{i} \text{Amortisierte-Kosten}_k$$

gilt.

Satz 9.1 *Wir machen die folgenden Annahmen:*

(a) $\textit{Amortisierte-Kosten}_k = \textit{Wirkliche-Kosten}_k + \Phi(k) - \Phi(k-1)$ *für alle* $k \in \mathbb{N}$,

(b) $\Phi(0) = 0$ *und*

(c) $\Phi(k) \geq 0$ *für alle* $k \in \mathbb{N}$.

Dann ist die amortisierte Laufzeit mindestens so groß wie die wirkliche Laufzeit, denn es gilt

$$\sum_{k=1}^{i} \textit{Amortisierte-Kosten}_k \geq \sum_{k=1}^{i} \textit{Wirkliche-Kosten}_k.$$

Beweis: Wir erhalten

$$\begin{aligned}
\sum_{k=1}^{i} \text{Amortisierte-Kosten}_k &= \left(\sum_{k=1}^{i} \text{Wirkliche-Kosten}_k\right) + \sum_{k=1}^{i} (\Phi(k) - \Phi(k-1)) \quad \text{wegen (a)} \\
&= \left(\sum_{k=1}^{i} \text{Wirkliche-Kosten}_k\right) + \Phi(i) - \Phi(0) \qquad \text{Teleskopsumme} \\
&= \left(\sum_{k=1}^{i} \text{Wirkliche-Kosten}_k\right) + \Phi(i) \qquad \text{wegen (b).} \\
&\geq \sum_{k=1}^{i} \text{Wirkliche-Kosten}_k \qquad \text{wegen (c).}
\end{aligned}$$

und das war zu zeigen. □

Aufgabe 75

(a) Gegeben sei ein Stack mit den Operationen *Push* (Hinzufügen eines Elementes) und *Pop* (Entfernen des zuletzt hinzugefügten Elementes). Eine neue Operation *Multipop(k)*, welche die obersten k Elemente entfernt, soll mit Hilfe der Pop-Operation implementiert werden. Analysiere mit Hilfe einer Potentialfunktion die Kosten von n Operationen, wenn der Stack zu Beginn leer ist.

(b) Gegeben sei ein Zähler mit einer Inkrement und einer Dekrement Operation. Analysiere die Worst-Case-Kosten einer Folge von n Operationen (der Zähler beginnt bei 0 und hat unbeschränkte Länge).

(c) Es soll eine Schlange implementiert werden. Dabei stehen als Operationen nur die Operationen eines Stacks zur Verfügung. Zeige, dass die Schlange auf zwei Stacks in konstanter amortisierter Laufzeit implementiert werden kann.

Aufgabe 76

Auf einem Array der Länge $n = 2^k$ soll die *Bit-Reversal-Permutation* ausgeführt werden, d.h. ein Element an Position i, wobei i die Binärdarstellung $(i_{k-1}, \ldots, i_0)$ besitzt, soll an Position $(i_0, \ldots, i_{k-1})$ gebracht werden. Zeige, dass die Bit-Reversal-Permutation in Zeit $O(n)$ ausgeführt werden kann. Der verwendete Rechner sei eine Registermaschine mit k-Bit Registern und direktem sowie indirektem Speicherzugriff. Als „arithmetische" Operationen sind nur zyklische Shifts um eine Position, sowie Operationen auf dem untersten Bit erlaubt. Hinweis: Verwende einen „umgedrehten" Zähler mit geringen amortisierten Kosten.

Beispiel 9.2 **Dynamische Hashtabelle**

In der Praxis lässt sich die Anzahl der durch Hashing einzufügenden Schlüssel nicht genau vorhersagen. Eine zu große Hashtabelle belegt unnötig Platz, eine zu kleine Hashtabelle reduziert den Laufzeitvorteil von Hashing durch viele Kollisionen. Wir beginnen mit einer leeren Hashtabelle der Größe $1 = 2^0$ und versuchen, die Größe der Hashtabelle dynamisch anzupassen. Zu

$$\lambda := \frac{\text{Anzahl Schlüssel}}{\text{Tabellengröße}}$$

wählt man zum Beispiel folgende Heuristik:

- Wenn $\lambda \geq 1$, füge sämtliche Schlüssel in eine neue Hashtabelle doppelter Größe ein.
- Wenn $\lambda \leq \frac{1}{4}$, füge sämtliche Schüssel in eine neue Hashtabelle der halben Größe ein.

In beiden Fällen ist nach der Reorganisation $\lambda = \frac{1}{2}$. Wir setzen idealisiert die wirklichen Kosten für eine Operation auf 1, wenn keine Reorganisation erforderlich ist. Wenn andererseits eine Reorganisation durchzuführen ist, dann setzen wir den wirklichen Preis für eine Operation auf m, falls die alte Tabelle die Größe m besitzt. Wir führen ein Buchhalter-Argument durch.

- Wenn die Hashtabelle mit der aktuellen Größe m zu klein wird, dann sind zwischenzeitlich mindestens $m/2$ Schritte ohne Reorganisation vergangen. Die durch die gerade durchgeführte Reorganisation verursachten Kosten von m ordnen wir den letzten $m/2$ Schritten zu. Die amortisierte Laufzeit für eine Operation ist damit höchstens $1+2=3$, da wir als wirkliche Kosten 1 angenommen haben.

- Wenn die Hashtabelle mit der aktuellen Größe m zu groß wird, so geschieht dies nach frühestens $m/4$ Schritten, wenn nämlich die ursprüngliche Auslastung von $1/2$ auf $1/4$ gesunken ist. Die Reorganisationskosten von m sind also auf die letzten $m/4$ Operationen zu verteilen und wir erhalten im Worst-Case die amortisierte Laufzeit $1+4=5$ für eine Operation.

Insgesamt gelingen also k Operationen in idealisierter Laufzeit höchstens $5k$.

Aufgabe 77
Binäre Suche in einem sortierten Array benötigt logarithmische Zeit, das Einfügen hingegen lineare Zeit. Betrachte daher folgende Datenstruktur zur Unterstützung der Suche- und Einfüge-Operationen: Sei n gegeben mit Binärdarstellung $(b_{k-1},\ldots,b_0)$. Es gebe k sortierte Arrays A_i der Länge 2^i, $i=0,\ldots,k-1$. Jedes Array ist entweder leer oder voll, entsprechend dem Wert von b_i. Daher enthalten die Arrays zusammen n Elemente. Jedes einzelne Array ist sortiert. Beschreibe eine Implementierung von Lookup() und von Insert() mit amortisierter Laufzeit $O(\log_2 n)$ für Insert. Bestimme die Worst-Case-Zeit der Lookup-Operation. Ist eine effiziente Implementierung von Delete() möglich?

9.2 Splay-Bäume

Wir betrachten Datenstrukturen für geordnete Wörterbücher und die zu unterstützenden Operationen

- Lookup(x): Entscheide, ob das Wort x im Wörterbuch ist.

- Insert(x): Füge das Wort x ins Wörterbuch ein.

- Delete(x): Lösche das Wort x aus dem Wörterbuch.

- Min(): Bestimme das Mininum des Wörterbuchs.

Viele Typen von Suchbäumen wie AVL-, B- und Brother-Bäume wurden zur Unterstützung dieser Operationen entwickelt [CLR]. Diese Datenstrukturen haben alle in etwa dieselben Eigenschaften:

(1) Logarithmische Worst-Case-Laufzeit für jede Operation.

(2) Hohe Konstanten in der asymptotischen Laufzeitanalyse, da es sich um balancierte Bäume handelt, und wir nach einer Operation die Balance-Forderungen wieder erfüllen müssen.

(3) Eine nicht unerhebliche Speicherkomplexität, da jeder Knoten des Baums Balance-Informationen für seinen Teilbaum speichert.

(4) Ein nicht-adaptives Verhalten der Baumstruktur, zum Beispiel werden häufig nachgefragte Schlüssel im Allgemeinen nicht an der Spitze des Baumes liegen.

Konventionelle Suchbäume sind den Splay-Bäumen (oder selbst-organisierten, binären Suchbäumen) in den Punkten (2), (3) und (4) deutlich unterlegen. Dies erreichen wir nur unter Aufgabe der Eigenschaft (1): Statt der Laufzeit $O(\log_2 n)$ pro Operation garantieren wir eine Laufzeit von $O(n \cdot \log_2 n)$ für n Operationen. Splay-Bäume können somit zwar ineffizient für individuelle Operationen sein, sie sind jedoch hervorragend im Bezug auf die *Gesamtlaufzeit* aller Operationen.

Die Architektur eines Splay-Baumes ist die eines binären Suchbaumes. Jeder Knoten hat maximal zwei Kinder, und wenn u im linken Teilbaum von v und w im rechten Teilbaum von v liegt, gilt:

$$\text{Schlüssel}(u) < \text{Schlüssel}(v) < \text{Schlüssel}(w).$$

Die Operationen Lookup(x), Insert(x) und Delete(x), sowie Min() führen wir auf die Splay-Operation zurück. In der Operation Splay(x) wird zunächst nach dem Schlüssel x gesucht und dabei das folgende Suchverfahren durchgeführt:

(1) Wir beginnnen an der Wurzel, vergleichen den Wert v der Wurzel mit x und gehen, falls v nicht der gesuchte Schlüssel ist, entweder in den linken Teilbaum ($x < v$) oder in den rechten Teilbaum ($x > v$).

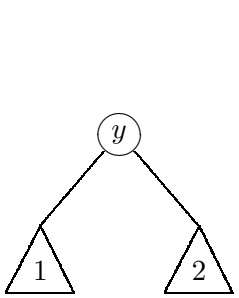
(2) Dieses Verfahren wird solange wiederholt bis der Schlüssel x gefunden wurde oder bis ausgeschlossen werden kann, dass sich x im Baum befindet.

Als Resultat dieser Suchoperation wird der größte Schlüssel y im Baum mit $y \leq x$, bzw. der kleinste Schlüssel y mit $y \geq x$ bestimmt.

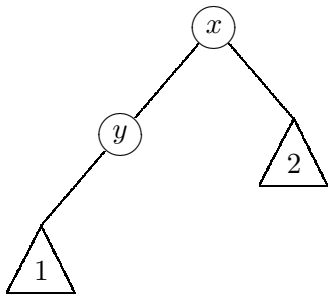
Der Knoten des gefundenen Schlüssels wird sodann in einer Aufwärtsphase zur Wurzel „rotiert“, um den möglicherweise langen Pfad zur Wurzel zu verkürzen. Bevor wir die Splay-Operation detailiert beschreiben und analysieren, führen wir die übrigen Operationen auf die Splay-Operation zurück.

- Lookup(x): Führe Splay(x) aus und vergleiche den Schlüssel an der Wurzel mit x.

- Insert(x): Führe Splay(x) durch und füge x als neue Wurzel ein. Wenn wir zum Beispiel annehmen, dass Splay(x) den größten Schlüssel y mit $y \leq x$ findet, modifizieren wir wie folgt:

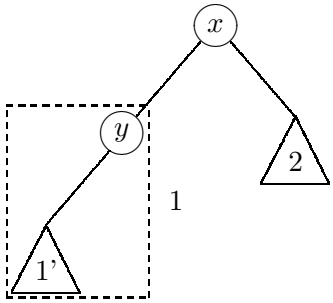


Nach Splay(x)

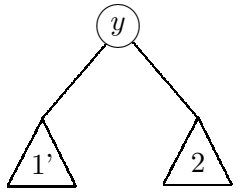


Neuer Baum nach Einfügen

- Delete(x): Führe Splay(x) durch. Seien 1 und 2 die Teilbäume der neuen Wurzel x. Führe Splay(x) auf dem linken Teilbaum 1 aus, dessen Schlüssel alle kleiner als x sind. Sei y die Wurzel und $1'$ dessen Unterbaum. Beachte, dass es im Teilbaum 1 keinen Schlüssel größer als y gibt und y daher keinen rechten Nachfahren besitzt. Wir löschen x und der neue Baum hat die Wurzel y mit den Unterbäumen $1'$ sowie 2.



Nach Splay(x) im ganzen Baum und Splay(x) im Teilbaum 1

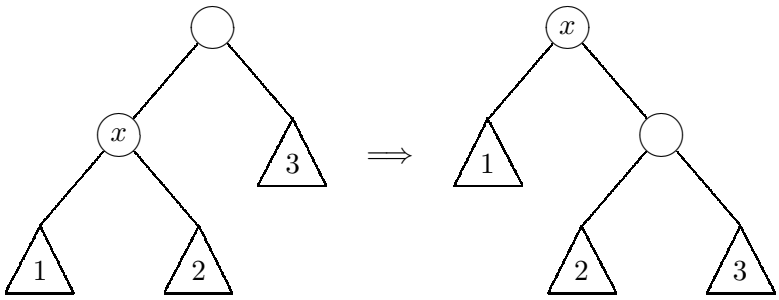


Neuer Baum nach Löschen von x

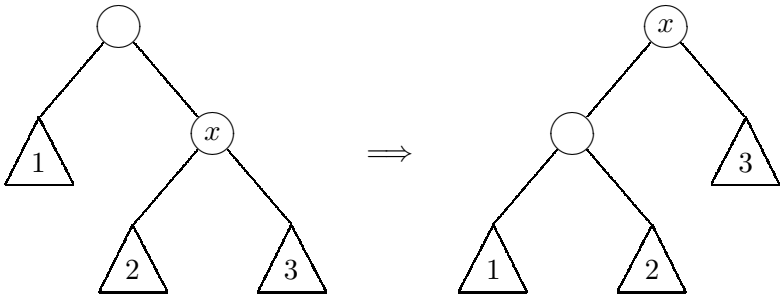
- Min(): Führe Splay($-\infty$) aus und gib den Schlüssel der Wurzel aus.

Jede Operation wird also mit höchstens zwei Splay-Operationen implementiert; allerdings sind konstant viele Kanten zusätzlich zu modifizieren, aber diese Zusatzkosten sind gegen die Splay-Operationen vernachlässigbar.

Wir implementieren jetzt die Aufwärtsphase der Splay-Operation mit einer Folge von Rechtsrotationen und Linksrotationen.



Rechtsrotation bei Splay-Bäumen.



Linksrotation bei Splay-Bäumen.

Algorithmus 9.2 **Die Splay-Operation.**

(1) Die Eingabe besteht aus einem binären Suchbaum T und dem Schlüssel x

(2) Suche nach x in der Abwärtsphase:

Suche den Schlüssel bzw. Knoten x: Wir beginnnen in der Wurzel, vergleichen den Wert v der Wurzel mit x und gehen, falls v nicht der gesuchte Schlüssel ist, entweder in den linken Teilbaum ($x < v$) oder in den rechten Teilbaum ($x > v$). Wenn $x = v$ wird die

Suche abgebrochen und ansonsten solange fortgesetzt, bis entweder ein Blatt ereicht wird oder der Schlüssel gefunden wird.

Sei y der gefundene Schlüssel.

(3) Die Aufwärtsphase bewegt y zur Wurzel.

WHILE (y steht nicht an der Wurzel) DO

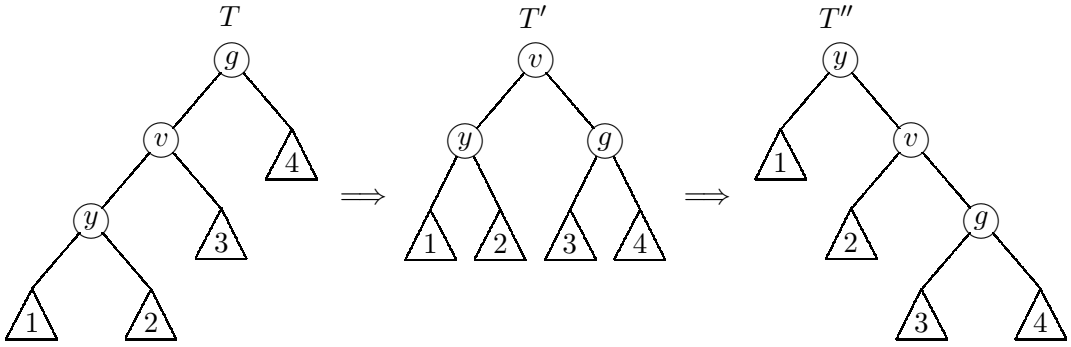
(3a) Sei v der Vater und g der Großvater von y (sofern dieser existiert).

(3b) Rotiere gemäß folgender Fallunterscheidung:

Zick-Zick: y und v sind beide linke oder beide rechte Kinder. Rotiere zuerst am Großvater g und dann am Vater v.

Zick-Zack: y ist ein linkes bzw. rechtes Kind und v ist ein rechtes bzw. linkes Kind. Wir rotieren zuerst am Vater v und dann am vorherigen Großvater g.

Zick: Der Vater v ist die Wurzel. Führe die entsprechende Rotation durch, um y an die Wurzel des Baumes zu bringen.

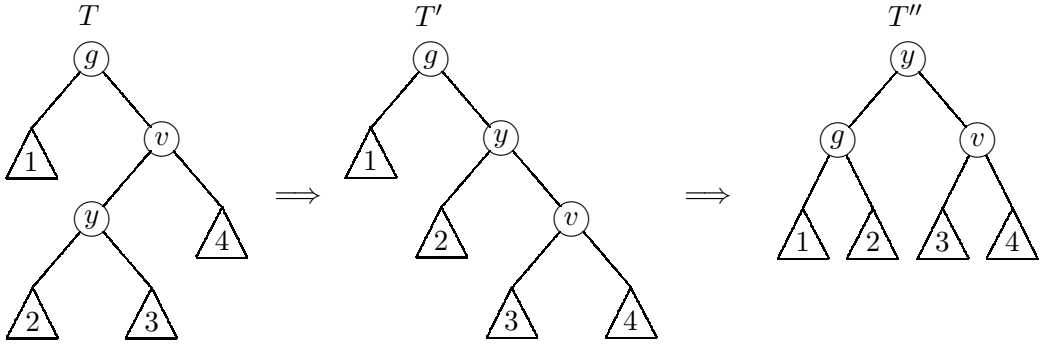


Der Zick-Zick Fall.

Was passiert im Zick-Zick Fall?

- y ist näher zur Wurzel gerückt.
- Alle Knoten in 1 und 2 sind ebenso der Wurzel näher gekommen.
- Alle Knoten in 3 oder 4 rücken von der Wurzel weg.

Wiederholte Anwendungen des Zick-Zick Falls führen also eine Art von Balancierung durch: y rückt der Wurzel um zwei Kanten näher, während alle Knoten in 1 oder 2 der Wurzel um mindestens eine Kante näherrücken, die Knoten der Bäume 3 und 4 nehmen in den nachfolgenden Anwendungen des Zick-Zick Falls an der Aufwärtsbewegung teil. Die Knoten des Suchpfades rücken der Wurzel somit fast um die Hälfte näher: Wir nennen dies die *Suchpfad-Eigenschaft.*

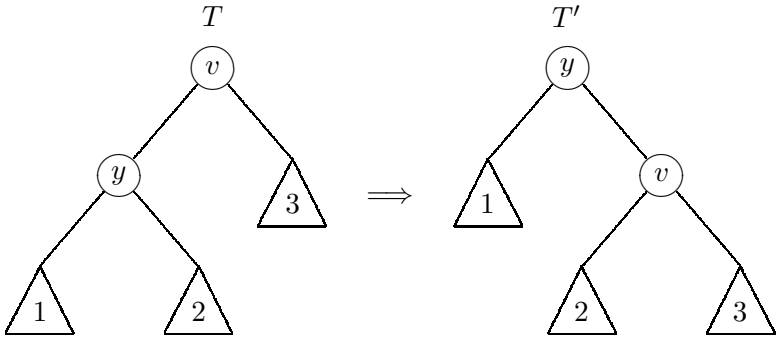


Der Zick-Zack Fall.

Was passiert im Zick-Zack Fall?

- y ist näher zur Wurzel gerückt.
- Alle Knoten in 2 und 3 sind ebenso der Wurzel nähergerückt.
- Die Knoten in 4 behalten ihren Abstand, die Knoten in 1 rücken von der Wurzel weg.

y rückt der Wurzel um zwei Kanten näher, während alle Knoten in 2 oder 3 der Wurzel um eine Kante näherrücken; die Knoten in 1 und 4 nehmen in nachfolgenden Anwendungen des Zick-Zack Falls an der Aufwärtsbewegung teil. Mit anderen Worten, nach wiederholter Anwendung des Zick-Zack Falls rücken die Knoten des Suchpfades der Wurzel auch diesmal fast um die Hälfte näher und die Suchpfad-Eigenschaft ist wiederum erfüllt.

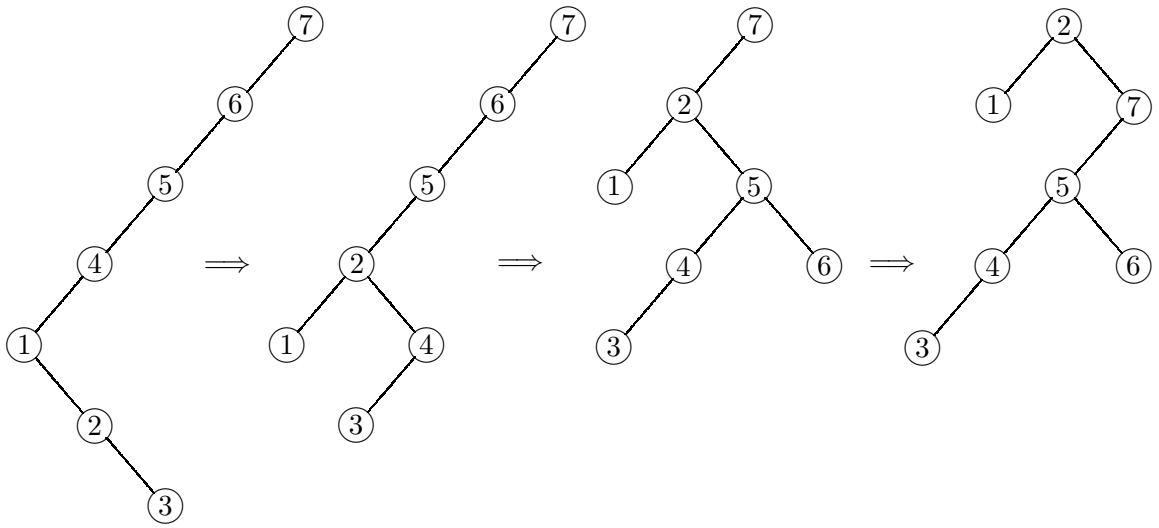


Der Zick Fall.

Aufgabe 78
Wir nehmen an, dass der Zick-Zick Fall genau so wie der Zick-Zack Fall ausgeführt wird. Konstruiere einen Splay-Baum T mit n Knoten und eine Folge von n Lookup-Operationen, so dass die modifizierten Splay-Operationen insgesamt die Laufzeit $\Omega(n^2)$ benötigen.

Das folgende Beispiel zur Splay-Operation zeigt die Herkunft der Bezeichung Splay-Bäume: „to splay“ ist der englische Ausdruck für „verbreitern“.

Beispiel 9.3



Wir rufen Splay(2) für den ersten Baum in der obigen Abbildung auf. Wir gehen den Baum herab zum Knoten 2.

- Der Knoten 2 und sein Vaterknoten 1 sind jeweils ein rechtes, bzw. linkes Kind. Da wir im Zick-Zack-Fall sind, rotieren wir zunächst am Vater 1 und anschließend am Großvater 4. Der zweite Baum der Abbildung zeigt das Resultat.
- Der Knoten 2 und sein Vaterknoten 5 sind beide jeweils linke Kinder: Im Zick-Zick-Fall rotieren wir zunächst am Großvater 6 und anschließend an 5. Der dritte Baum der Abbildung zeigt das Resultat.
- Der Vaterknoten 7 von 2 ist die Wurzel: Für diesen Zick-Fall rotieren wir am Vater 7, um 2 an die Wurzel des Baumes zu bringen. Der vierte Baum der Abbildung zeigt das Resultat.

Der gesuchte Wert 2 steht nun an der Wurzel des Suchbaumes.

Wir beschreiben und analysieren im Folgenden die Aufwärtsphase im Detail. Zur Betrachtung der amortisierten Laufzeit definieren wir eine Potentialfunktion Φ für binäre Suchbäume.

Definition 9.3 Sei T ein binärer Suchbaum.

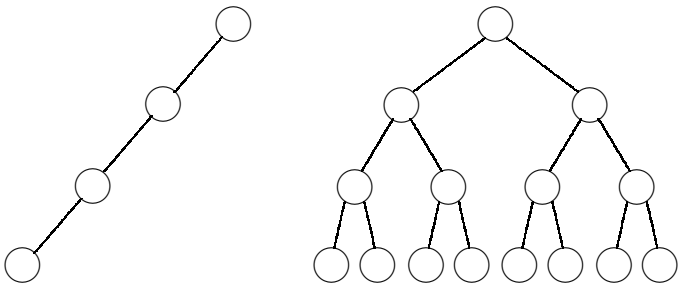
(a) Für einen Knoten w sei $|T(w)|$ die Knotenanzahl im Teilbaum mit Wurzel w (inklusive w).

(b) Für einen Knoten w setzen wir den Rang $R_T(w) := \log_2 (|T(w)|)$ und definieren die Potentialfunktion

$$\Phi(T) := \sum_{w \in T} R_T(w)$$

als die Summe aller Ränge der Knoten.

Betrachten wir zwei Beispiele zur Potentialfunktion.



Potential $\Theta(n \log_2 n)$ Potential $\Theta(n)$

Der Baum T habe n Knoten. Falls der Baum die Form einer Liste hat, gilt:

$$\Phi(T) = \sum_{i=1}^{n} \log_2 i = \log_2 n! = \Theta(n \log_2 n).$$

Um den kleinsten Schlüssel zu finden, sind n Schritte nötig. Falls der Baum T hingegen ein vollständiger binärer Baum mit $n = 2^k - 1$ Schlüsseln ist, sind logarithmisch viele Schritte ausreichend. In diesem Fall ist das Potential durch

$$\Phi(T) = \sum_{i=1}^{k} 2^{k-i} \cdot \log \left(2^i - 1\right) \leq n \sum_{i=0}^{\infty} \frac{i}{2^i} = O(n).$$

gegeben.

Wir zeigen, dass die Laufzeit einer Splay-Operation auf einem Baum mit n Knoten in amortisierter Laufzeit $O(\log_2 n)$ gelingt. Da die Laufzeit der Aufwärtsphase die der Abwärtsphase dominiert, betrachten wir nur die Aufwärtsphase. Wir bezeichnen mit T_i den Baum nach der i-ten Rotation der Splay-Operation. Unsere Analyse unterscheidet die drei Fälle der Splay-Operation.

Der Zick-Zick-Fall: y und der Vater v von x sind beide linke oder beide rechte Kinder. Rotiere zuerst am Großvater g von x und dann am Vater v.

Das Ziel dieser Rotation ist nicht nur das Heraufbringen von y, sondern auch die Kontraktion des bisher gelaufenen Weges. Zwar scheint dieses nicht stattzufinden (v und g kommen der Wurzel nicht näher), aber v und g sind jetzt Nachfahren von y und nachfolgende Zick-Zick-Operationen verkürzen den Abstand von v und g zur Wurzel.

Sei T der binäre Suchbaum vor Ausführung der Zick-Zick-Operation und T'' der binäre Suchbaum nach der Ausführung. Wir setzen die wirklichen Kosten der Rotation auf 1 und erhalten deshalb

$$\text{Amortisierte-Kosten(Zick-Zick)} = 1 + \Phi(T'') - \Phi(T).$$

Wir beachten, dass sich die Teilbaumgröße nur für y, v und g ändert. Es gilt also

$$\Phi(T'') - \Phi(T) = [R_{T''}(y) - R_T(y)] + [R_{T''}(v) - R_T(v)] + [R_{T''}(g) - R_T(g)]\,. \tag{9.1}$$

Wegen $R_T(y) < R_T(v)$ und $R_{T''}(v) < R_{T''}(y)$ gilt

$$R_{T''}(v) - R_T(v) < R_{T''}(y) - R_T(y).$$

Als Konsequenz erhalten wir aus (9.1)

$$\Phi(T'') - \Phi(T) < 2 \cdot [R_{T''}(y) - R_T(y)] + R_{T''}(g) - R_T(g).$$

Unser Ziel ist der Nachweis von

$$R_{T''}(g) - R_T(g) \overset{!}{\leq} R_{T''}(y) - R_T(y) - 1, \tag{9.2}$$

denn dies impliziert

$$\Phi(T'') - \Phi(T) + 1 \leq 3 \cdot [R_{T''}(y) - R_T(y)]\,.$$

Die linke Seite ist die amortisierte Laufzeit einer Zick-Zick-Operation und wir erhalten

$$\text{Amortisierte-Kosten(Zick-Zick)} \leq 3 \cdot [R_{T''}(y) - R_T(y)]\,. \tag{9.3}$$

Wir müssen noch Ungleichung (9.2) beweisen. Zuerst beachten wir, dass für den mittleren Baum T' der Zick-Zick-Operation

$$\begin{aligned} |T'(v)| &= |T'(y)| + |T'(g)| + 1 \\ &> 2 \cdot \min\left\{T'(y)|, |T'(g)|\right\}. \end{aligned}$$

gilt. Durch Logarithmieren erhalten wir:

$$R_{T'}(v) > 1 + \min\left\{R_{T'}(y), R_{T'}(g)\right\}.$$

Da $R_{T'}(y) = R_T(y)$, $R_{T'}(v) = R_{T''}(y) = R_T(g)$ sowie $R_{T'}(g) = R_{T''}(g)$ folgt

$$R_{T''}(y) = R_T(g) > \min\{R_T(y), R_{T''}(g)\} + 1.$$

Wir unterscheiden zwei Fälle:
Fall 1: Es gilt $R_T(y) \leq R_{T''}(g)$. Somit ist $R_{T''}(y) > 1 + R_T(y)$ und wir erhalten wegen $R_T(y) \leq R_{T''}(g)$,

$$R_{T''}(y) - R_T(y) - 1 > 0 > R_{T''}(g) - R_T(g).$$

Fall 2: Es gilt $R_T(y) > R_{T''}(g)$ und somit $R_T(g) > 1 + R_{T''}(g)$. Da stets $R_{T''}(y) - R_T(y) > 0$ ist, folgt

$$R_{T''}(g) - R_T(g) < -1 < R_{T''}(y) - R_T(y) - 1.$$

Wir haben die Ungleichung (9.2), also $R_{T''}(g) - R_T(g) \leq R_{T''}(y) - R_T(y) - 1$, bewiesen.

Der Zick-Zack-Fall: y ist ein linkes (bzw. rechtes) Kind und v ist ein rechtes (linkes) Kind. Wir rotieren zuerst am Vater und dann am vorherigen Großvater. Mit analogen Argumenten zeigt man wie im Zick-Zick-Fall, dass die amortisierte Laufzeit einer Zick-Zack-Operation durch

$$\text{Amortisierte-Kosten(Zick-Zack)} \leq 3 \cdot \left[R_{T_i}(y) - R_{T_{i-1}}(y)\right] - 1 \tag{9.4}$$

nach oben beschränkt ist.

Der Zick-Fall: : Der Vater v von y ist die Wurzel. Führe die entsprechende Rotation durch, um y an die Wurzel des Baumes zu bringen. In diesem Fall ist die amortisierte Laufzeit der Einzelrotation

$$1 + R_{T'}(y) - R_T(y) + R_{T'}(v) - R_T(v),$$

denn wir haben die wirkliche Operation der Einzelrotation gezählt wie auch die Veränderung der Potentialfunktion vermerkt. (Eine Änderung findet nur für die Knoten y und v statt). Wegen $R_{T'}(y) = R_T(v)$ und $R_{T'}(v) \leq R_{T'}(y)$ gilt:

$$\begin{aligned}
\text{Amortisierte-Kosten(Zick)} &= 1 + [R_{T'}(v) - R_T(v)] + [R_{T'}(y) - R_T(y)] \\
&= 1 + R_{T'}(v) - R_T(y) \\
&< 1 + R_{T'}(y) - R_T(y) \\
&\leq 1 + 3 \cdot [R_{T''}(y) - R_T(y)]\,.
\end{aligned} \tag{9.5}$$

Wir kennen obere Schranken Amortisierte-Kosten$_i$ für die Einzeloperationen (siehe (9.3), (9.4) und (9.5)). Wenn wir k Einzeloperationen durchführen, sind die ersten $k-1$ Zick-Zack- und Zick-Zick-Fälle und der letzte ist ein Zick-Fall. Also erhalten wir, wenn T_i der Baum nach den $i$ten Rotationen ist,

$$\begin{aligned}
\sum_{i=1}^{k} \text{Amortisierte-Kosten}_i &\leq \sum_{i=1}^{k-1} \text{Amortisierte-Kosten}_i + \text{Amortisierte-Kosten}_k \\
&\leq \sum_{i=1}^{k-1} 3 \cdot [R_{T_i}(y) - R_{T_{i-1}}(y)] + 3 \cdot [R_{T_k}(y) - R_{T_{k-1}}(y)] + 1 \\
&= 3 \cdot [R_{T_k}(y) - R_{T_0}(y)] + 1 \\
&\leq 3 \cdot R_{T_k}(y) + 1
\end{aligned}$$

als obere Schranke für die amortisierte Laufzeit der vollständigen Splay-Operation. Wenn der Baum T_k maximal n Knoten hat, gilt

$$\text{Amortisierte-Kosten(Splay)} \leq 3 \cdot \log_2 n + 1. \tag{9.6}$$

wegen $R_{T_k}(y) = \log_2(\ |T(y)|\ )$. Da nach Satz 9.1 amortisierte Laufzeiten obere Schranken der wirklichen Laufzeiten sind, erhalten wir

Satz 9.4 *Auf einen am Anfang leeren Splay-Bäume kann man n Operationen des Typs Lookup(x), Insert(x), Delete(x) und Min() in Zeit $O(n \cdot \log_2 n)$ ausführen.*

Beweis: Wir haben gezeigt, dass die Operationen „Lookup“ und „Min“ identisch mit Splay-Operationen sind. Die Insert- und die Delete-Operation bestehen aus einer bzw. zwei Splay-Operationen und zusätzlichen $O(1)$ vielen Schritten.

Die amortisierte Laufzeit einer Splay-Operation in einem Splay-Baum mit n Knoten ist durch $O(\log_2 n)$ beschränkt. Aus Satz 9.1 folgt deshalb die Behauptung. □

Wie gut sind Splay-Bäume im Vergleich zu beliebigen anderen Algorithmen? Wir untersuchen diese zentrale Frage für Folgen σ von Lookup-Operationen und lassen Splay-Bäume sogar gegen die folgende Klasse von off-line Algorithmen antreten:

> Sei T ein binärer Suchbaum für die Schlüsselmenge $S = \{1, \ldots, n\}$ und $\sigma = (\sigma_1, \ldots, \sigma_m) \in S^m$ beschreibe eine Folge von Lookup-Anfragen. Ein off-line Algorithmus A kann die gesamte Folge σ zu Beginn der Berechnung einsehen.
>
> A arbeitet in Phasen. Für Phase 1 wird $T^0 = T$ und $i = 1$ gesetzt. In Phase i beginnt A mit dem Baum T^{i-1} und sucht zuerst nach σ_i, indem der Suchpfad in T^{i-1} durchlaufen wird. Dann darf A eine Menge von r_i Rotationen an beliebiger Stelle in T^{i-1} ausführen: Der entstehende Baum sei T^i. Wenn l_i die Länge des Suchpfads für σ_i ist, dann definieren wir
>
> $$\text{Kosten}_A(T, \sigma) = \sum_{i=1}^{m} (l_i + r_i)$$
>
> als die Kosten von A für die Folge σ mit anfänglichem Baum T.

Wir beachten zuerst, dass der Wettbewerbsfaktor der Splay-Bäume nach n Lookup-Operationen höchstens $O(\log_2 n)$ beträgt. Diese Beobachtung ist eine unmittelbare Konsequenz von Satz 9.4, da Splay-Bäume höchstens die Zeit $O(n \log_2 n)$ benötigen, während jeder andere Algorithmus mindestens $\Omega(n)$ Schritte ausführen muss, da ja n Lookup-Operationen zu beantworten sind. Tätsächlich wird in der „Dynamic Optimality Conjecture“ sogar vermutet, dass Splay-Bäume einen konstanten Wettbewerbsfaktor besitzen!

Offenes Problem 4

Die **Dynamic Optimality Conjecture** besagt, dass für jeden off-line Algorithmus A, für jeden binären Suchbaum T und jede Folge σ von m Lookup-Operationen auf T

$$\text{Kosten}_{\text{Splay}}(T, \sigma) = O(\text{Kosten}_A(T, \sigma) + m)$$

gilt. Er wird also angenommen, dass Splay-Bäume einen konstanten Wettbewerbsfaktor besitzen.

Die Dynamic Optimality Conjecture ist seit über 20 Jahren offen und konnte bisher nur für sehr eingeschränkte Folgen σ bestätigt werden. Zum Beispiel, wenn T die Schlüssel $1, \ldots, n$ so

abspeichert, dass ein Inorder-Durchlauf die Schlüssel in sortierter Reihenfolge ausgibt, dann gilt $\text{Kosten}_{\text{Splay}}(T, \sigma) = O(n)$ für die Reihenfolge $\sigma = (1, \ldots, n)$. (Beachte, dass das Ergebnis $\text{Kosten}_{\text{Splay}}(T, \sigma) = \omega(n)$ die Dynamic Optimality Conjekture widerlegt hätte. Warum?)

Die Dynamic Optimality Conjecture kann auch für beliebige Bäume T und Folgen σ bestätigt werden, wenn off-line-Algorithmen keine Rotationen ausführen dürfen. Wir können ein verwandtes Ergebnis sogar mit unseren Methoden herleiten: Sei p_i die Wahrscheinlichkeit, dass Schlüssel i Argument einer Lookup-Anfrage ist, und es gelte $\sum_{i=1}^{n} p_i = 1$. Dann heißt ein binärer Suchbaum *optimal* für die Verteilung p, wenn die erwartete Suchzeit

$$s = \sum_{i=1}^{n} p_i \cdot \text{Tiefe}\,(i\text{-ter Schlüssel})$$

minimal ist. In der nächsten Übungsaufgabe zeigen wir, dass Splay-Bäume selbst dann eine bis auf einen konstanten Faktor optimale erwartete Suchzeit erreichen, wenn sie mit einem beliebig schlechten Baum beginnen müssen. Mit anderen Worten, Splay-Bäume passen sich den Daten fast optimal an!

Aufgabe 79

Es sei ein beliebiger, statischer, binärer Suchbaum S mit n Schlüsseln gegeben. T sei ein weiterer beliebiger binärer Suchbaum mit derselben Schüsselmenge. Zeige, dass sich jede beliebige Folge von Suchoperationen, die auf S in m Schritten abgearbeitet wird, mit den Splay-Baumoperationen in $O(m + n^2)$ Schritten abarbeiten lässt, wenn mit T begonnen wird.

Hinweis: Betrachte die Analyse der amortisierten Laufzeit der Splay-Operation. Es sei jeder Knoten im Baum mit einem positiven Gewicht belegt. Ersetze $|T(w)|$ in der Definition der Potentialfunktion durch die Summe der Gewichte über alle Knoten im Teilbaum von w.

Derselbe Beweis wie in der Vorlesung zeigt, dass die amortisierte Laufzeit von Splay durch $3 \cdot (\log |T(\text{wurzel})| - \log |T(x)|) + 1$ beschränkt ist. Gib dem Knoten, der Schlüssel x speichert, das Gewicht 3^{d-d_x}, wenn x in S in Tiefe d_x gespeichert ist, und d die Tiefe von S ist.

Aufgabe 80

Es sei eine Wahrscheinlichkeitsverteilung $(p_1, \ldots, p_n)$ auf einer Schlüsselmenge gegeben. Ein optimaler binärer Suchbaum ist ein binärer Suchbaum für die Schüsselmenge, der die erwarteten Kosten $\sum_{i=1}^{n} p_i \cdot Tiefe(i)$ einer Suchoperation minimiert. $Tiefe(i)$ ist die Tiefe des Knotens, der den $i$ten Schlüssel x_i speichert.

Gib einen effizienten Algorithmus an, der zu einer gegebenen Verteilung einen optimalen Suchbaum konstruiert.

Aufgabe 81

Ein *Move-To-Root-Baum* ist ein geordneter binärer Suchbaum, bei dem das zuletzt gesuchte Element ähnlich wie bei Splay-Bäumen an die Wurzel gebracht wird, allerdings werden nur einfache Rotationen am Vaterknoten verwendet wie im Zick-Fall bei Splay-Bäumen. Konstruiere einen binären Suchbaum mit n Elementen sowie eine Zugriffsfolge der Länge n, so dass die Move-To-Root Strategie $\Omega(n^2)$ Schritte benötigt.

Aufgabe 82

Bestimme die amortisierte Laufzeit der Zick-Zack Operation.

Aufgabe 83

(a) Beschreibe einen Algorithmus, der die Operation Intervall(a, b) auf einem Splay-Baum T durchführt. Die Operation liefert alle Schlüssel y von T mit $a \leq y \leq b$ in aufsteigender Reihenfolge. Der Algorithmus soll in Worst-Case-Zeit $O(\text{Tiefe}(T) + Y)$ laufen, wobei Y die Anzahl der auszugebenden Schlüssel ist.

(b) Beschreibe einen Algorithmus, der Intervall(a, b) in amortisierter Zeit $O(\log_2 n + Y)$ auf einem Splay-Baum T der Größe n ausführt.

9.3 Binomische Heaps und Fibonacci-Heaps

Unser Ziel ist die Entwicklung von Datenstrukturen, um fundamentale Algorithmen wie Prim's Algorithmus für minimale Spannbäume und Dijkstra's Algorithmus für kürzeste Wege perfekt zu unterstützen.

Prim's Algorithmus vergrößert einen minimalen Spannbaum durch Hinzunahme eines Knotens, der mit einem Knoten des Spannbaumes durch eine Kante minimalen Gewichts verbunden ist. Nach Wahl des Knotens muss untersucht werden, ob seine Nachbarn jetzt kürzere Verbindungen zum neuen Spannbaum haben. Eine Datenstruktur für Prim's Algorithmus muss dazu folgende Operationen unterstützen:

- Insert(x): Füge den Schlüssel x ein.
- DeleteMin(): Finde minimalen Schlüssel und entferne diesen.
- DecreaseKey(x, Δ): Der Wert des Schlüssels x wird um Δ vermindert.

Diese Operationen sind auch ausreichend zur Implementierung von Dijkstra's Algorithmus. Beachte, dass die Lookup-Operation nicht unterstützt werden muss. Der Operation DecreaseKey wird auch nicht der Schlüssel x übergeben, sondern die Adresse von x, so dass nicht nach x gesucht werden muss.

Sei $G = (V, E)$ eine Eingabe für Prim's Algorithmus. Die einzelnen Operationen besitzen dann (wie auch im Fall von Dijkstra's Algorithmus) die folgenden Häufigkeiten,

- Insert: $O(|V|)$.
- DeleteMin: $O(|V|)$.
- DecreaseKey: $O(|E|)$.

Wir benötigen eine Datenstruktur, die die besonders häufigen DecreaseKey-Operationen effizient unterstützt. Für diesen Zweck werden wir *Fibonacci-Heaps* mit den folgenden Leistungsdaten für n Schlüssel entwickeln:

- Insert, DecreaseKey: Amortisierte Laufzeit $O(1)$.
- Delete, DeleteMin: Amortisierte Laufzeit $O(\log_2 n)$.

Natürlich ist die Worst-Case-Laufzeit einer *einzelnen* Datenstruktur-Operationen nicht wesentlich, sondern nur die Worst-Case-Laufzeit zur Ausführung *aller* Datenstruktur-Operationen. Aus der Laufzeit der Operationen für Fibonacci-Heaps und der Anzahl der benötigten Operationen folgt deshalb:

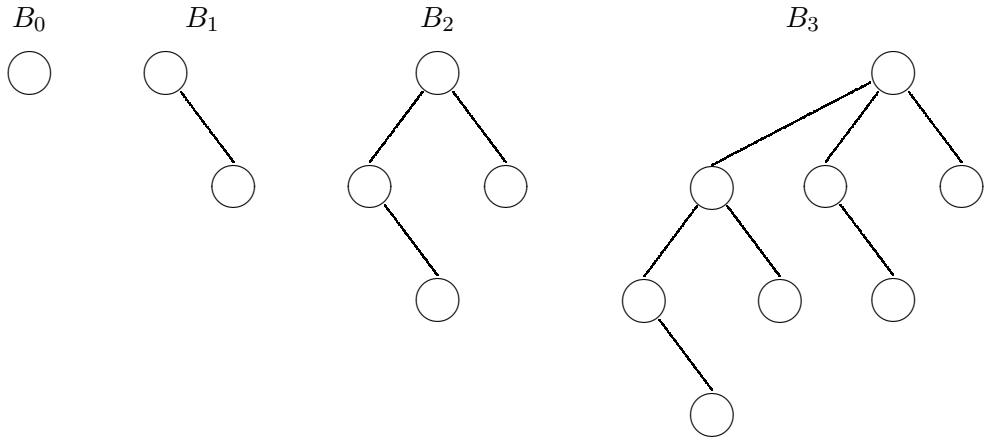
Satz 9.5 *Dijkstra's Algorithmus für kürzeste Wege und Prim's Algorithmus für minimale Spannbäume haben für Graphen $G = (V, E)$ die Laufzeit $O(|V| \cdot \log_2 |V| + |E|)$. Insbesondere ist die Laufzeit linear für $|E| = \Omega(|V| \cdot \log_2 |V|)$.*

Wir definieren zunächst binomische Bäume und binomische Heaps, um die Definition von Fibonacci-Heaps vorzubereiten:

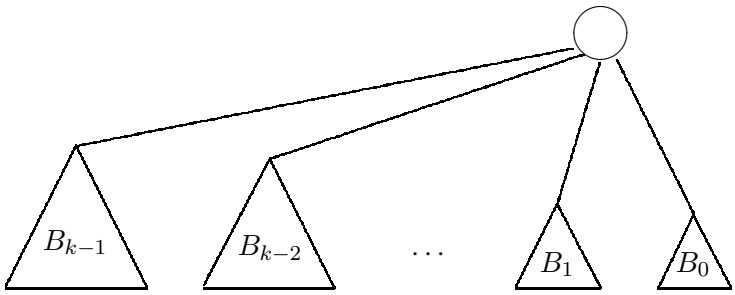
Definition 9.6 Wir definieren binomische Bäume B_k ($k \geq 0$) rekursiv.

- B_0 besteht nur aus einem Knoten.

- B_{k+1} entsteht aus zwei binomischen Bäumen B_k, indem eine Wurzel zum Kind der anderen Wurzel gemacht wird.



sind also die ersten vier binomischen Bäume. Man überlege sich, dass binomische Bäume die folgende Darstellung besitzen:



Weitere elementare Eigenschaften binomischer Bäume beweisen wir in dem folgenden Lemma.

Lemma 9.7 *Binomische Bäume B_k haben für $k \geq 0$ die folgenden Eigenschaften:*

(a) B_k hat Tiefe k.

(b) Die Wurzel von B_k hat k Kinder und die übrigen Knoten haben weniger als k Kinder.

(c) Es gibt in B_k genau $\binom{k}{i}$ Knoten der Tiefe i.

Beweis: Die drei Aussagen zeigt man durch Induktion über k. Wir beobachten für (a), dass B_{k+1} aus zwei binomischen Bäumen B_k, die um eine Tiefe verschoben sind, gebildet wird. Zum Nachweis von (b) ist nur zu beobachten, dass allein die Wurzel von B_{k+1} im Vergleich zu den Knoten von B_k im Grad anwächst. Für die dritte Behauptung beachte im Induktionsschritt, dass die Anzahl der Knoten der Tiefe i genau

$$\binom{k}{i} + \binom{k}{i-1} = \binom{k+1}{i}$$

beträgt und die Behauptung folgt. □

Insbesondere hat der binomische Baum B_k genau 2^k Knoten, und für $k \geq 3$ ist B_k kein binärer Baum. Binomische Bäume bilden die Basis binomischer Heaps.

Definition 9.8 Ein binomischer Heap ist eine Menge binomischer Bäume mit den folgenden Eigenschaften

(a) Für jedes i gibt es höchstens einen Baum B_i.

(b) Jeder Knoten eines binomischen Heaps speichert einen Schlüssel.

(c) Heap-Ordnung: Der Schlüssel des Vaters ist stets kleiner oder gleich den Schlüsseln seiner Kinder.

Warum fordern wir, dass es höchstens einen Baum B_i gibt? Sonst könnten alle Schlüssel als n Bäume des Typs B_0 gespeichert werden und die DeleteMin-Operation benötigte lineare Laufzeit zur Bestimmung des Minimums!

Wir beschreiben das Einfügen eines Schlüssels in einen binomischen Heap.

Algorithmus 9.9 **Einfügen in einen binomischen Heap**

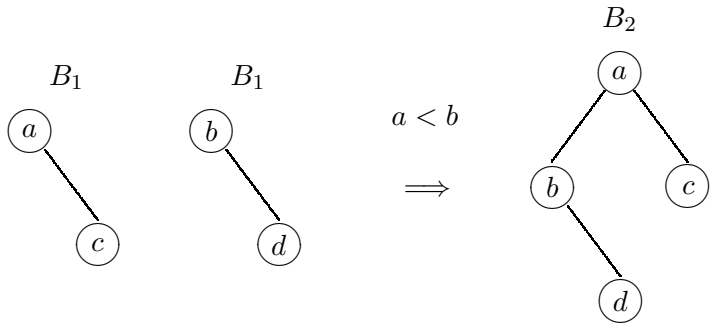
(1) Ein binomischer Heap H sei gegeben. Der Schlüssel x ist in H einzufügen.

(2) Generiere eine Kopie von B_0 und weise der Wurzel den Schlüssel x zu. Setze $k := 0$.

(3) WHILE (es gibt zwei Kopien von B_k) DO

(3a) Vereinige die beiden Kopien von B_k zu einer Kopie von B_{k+1}: Bestimme das Minimum der Werte der beiden Wurzeln und hänge den Baum mit größerem Wert an die Wurzel des anderen.

(3b) Setze $k := k + 1$.



Vereinigung zweier Binomischer Bäume beim Einfügen.

Wenn ein binomischer Heap insgesamt m Schlüssel speichert, dann gibt es genau dann einen Baum B_i in H, wenn das $i$te Bit von m gleich 1 ist. Es werden also genau so viele Vereinigungen während einer Einfüge-Operation durchgeführt, wie es aufeinanderfolgende Einsen in der Binärdarstellung vom m gibt (beginnnend mit dem niedrigstwertigen Bit). Damit kann also das Laufzeitverhalten des Einfügens durch die Laufzeit eines binären Zählers wiedergegeben werden!

Algorithmus 9.9 implementiert somit die Insert-Operation in konstanter, amortisierter Laufzeit, *falls* keine Delete-Operationen ausgeführt werden, denn die konstante, amortisierte Laufzeit folgt aus unserem Ergebnis über den binären Zähler. Aber die DeleteMin-Operation macht scheinbar alle Hoffnungen zunichte, da sie die amortisierte Laufzeit einer Insert-Operation erhöhen kann:

Füge $n = 2^k$ Schlüssel ein und der binomische Heap besteht zu diesem Zeitpunkt nur aus einem binomischen Baum B_k. Jetzt wechsle DeleteMin- und Insert-Operationen ab: Die DeleteMin-Operation entfernt die Wurzel und erzeugt die binomischen Bäume $B_0, B_1, \ldots, B_{k-1}$, die Insert-Funktion führt k Vereinigungen durch und liefert wieder den binomischen Baum B_k.

Wir erhalten aber in diesem Beispiel wie auch im allgemeinen Fall konstante amortisierte Laufzeit für die Insert-Operation, wenn wir der DeleteMin-Operation $k = \log_2 n$ Kosten zuordnen.

Wie sieht eine Implementierung der DecreaseKey-Operation mit konstanter amortisierter Laufzeit aus? Ein erster Versuch:

1. Setze den Wert des Schlüssels herab.
2. Wenn die Heap-Ordnung verletzt ist, trenne den Knoten des Schlüssels von seinem Vater.

Die Laufzeit ist $O(1)$, aber wir verletzen durch die Abtrennung die Definition binomischer Heaps, denn beliebige Teilbäume binomischer Bäume treten als neue Bäume auf. Wir geben daher die Struktur der binomischen Heaps auf und verwenden Fibonacci-Heaps, eine Verallgemeinerung von binomischen Heaps.

Eine Eigenschaft der Fibonacci-Heaps basiert auf den Fibonacci-Zahlen F_k, die der Datenstruktur auch den Namen gegeben haben:

$$F_k := \begin{cases} 0 & \text{falls } k = 0 \\ 1 & \text{falls } k = 1 \\ F_{k-1} + F_{k-2} & \text{falls } k \geq 2. \end{cases}$$

Die ersten Fibonacci-Zahlen sind:

k	0	1	2	3	4	5	6	7	8	9	10
F_k	0	1	1	2	3	5	8	13	21	34	55

Durch Induktion zeigt man, dass die Fibonacci-Zahlen die Eigenschaften

$$F_{k+2} \quad = 1 + \textstyle\sum_{i=1}^{k} F_i \quad \text{und} \quad F_{k+2} \quad \geq \phi^k \quad \text{mit } \phi := \frac{1}{2}\left(1 + \sqrt{5}\right) \approx 1,62 \tag{9.7}$$

für $k \geq 0$ besitzen. Der Wert ϕ heißt *goldener Schnitt*.

Definition 9.10 Ein Fibonacci-Heap ist eine Menge von Bäumen mit den folgenden Eigenschaften:

(a) Fibonacci-Eigenschaft: Jeder Knoten vom Grad k hat mindestens F_{k+2} Nachfahren, sich selbst mitgezählt.

Bäume mit der Fibonacci-Eigenschaft heißen Fibonacci-Bäume.

(b) Heap-Ordnung: Der Schlüssel des Vaters ist stets kleiner oder gleich den Schlüsseln seiner Kinder.

Wir verlangen auch, dass ein min-Zeiger auf den kleinsten Schlüssel zeigt.

Wir ändern unsere erste Implementierung der Decrease-Key Operation geringfügig.

Algorithmus 9.11 **Die DecreaseKey Operation für Fibonacci-Heaps**

(1) Ein Fibonacci-Heap H ist gegeben und ein Zeiger auf Schlüssel x, dessen Wert um Δ vermindert werden soll.

(2) Setze den Wert des Schlüssels x um Δ herab.

Wenn die Heap-Ordnung verletzt ist, trenne den Knoten des Schlüssels x von seinem Vater v.

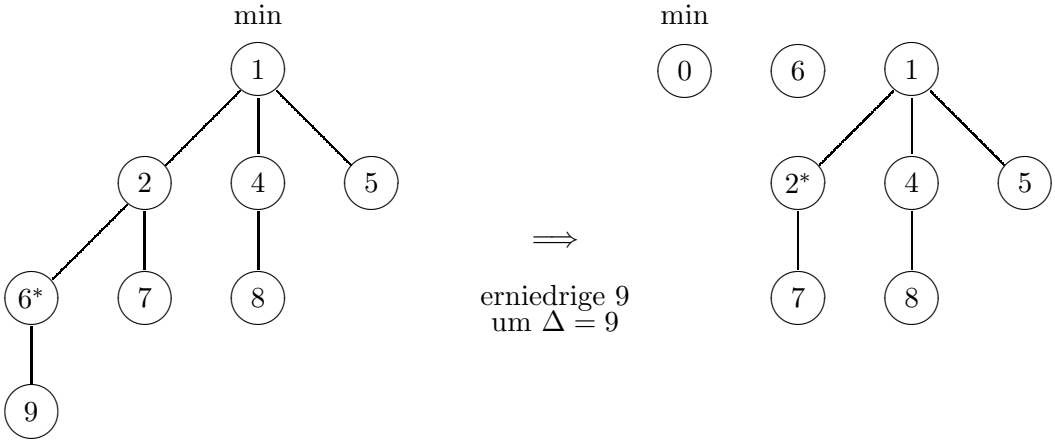
Wenn der Vater v sein zweites Kind verloren hat, trenne v von seinem Vater g. Falls g sein zweites Kind verloren hat, wiederhole diesen Schritt rekursiv für g und dessen Vaterknoten.

Kommentar: Um zu erkennen, ob ein Knoten sein zweites Kind verloren hat, führe eine Markierung für jedem Knoten ein.

(3) Überprüfe, ob der min-Zeiger auf x zeigen muss.

In Schritt (2) verwenden wir „*cascading Cuts*“, um die Fibonacci-Eigenschaft zu erzwingen: Wenn ein Knoten v sein zweites Kind verloren hat, dann verliert der Vater g von v möglicherweise zu viele Nachfahren. Das Abhängen von v reduziert den Grad seines Vaters g und wird, wie wir später sehen werden, die Fibonaci-Eigenschaft des Vaters retten.

Beispiel 9.4 Wir betrachten ein Beispiel zur DecreaseKey-Operation für Fibonacci-Heaps.



Die DecreaseKey-Operation für Fibonacci-Heaps

Die Abbildung zeigt einen Fibonacci-Heap, der nur aus einem Baum besteht. Der Knoten 6 ist markiert, d.h. er hat bereits ein Kind verloren. Wir wollen den Knoten 9 um $\Delta = 9$ erniedrigen. Die Heap-Ordung ist verletzt und wir trennen den Knoten vom Baum. Da der Vater, Knoten 6, bereits ein Kind verloren hat, trennen wir diesen ebenfalls ab. Dessen Vater, Knoten 2, ist noch nicht markiert. Schließlich aktualisieren wir den Minimum-Zeiger.

Die DecreaseKey-Operation „zerstückelt“ den Heap, und der DeleteMin-Operation fällt die Aufgabe zu, den Heap wieder zu reparieren. Wenn DeleteMin sowieso schon die Aufgabe des Groß-Reinemachens zukommt, dann liegt es auch nahe, das Einfügen eines Schlüssels durch die Hinzunahme eines Ein-Knoten-Baums zu implementieren.

Algorithmus 9.12 **Die Insert-Operation für Fibonacci-Heaps**

(1) Der Schlüssel x ist in einen Fibonacci-Heap einzufügen.

(2) Ein Baum, bestehend aus einem Knoten, wird erzeugt und der zu speichernde Schlüssel x wird in diesem Knoten gespeichert.

(3) Aktualisiere den min-Zeiger.

Damit ist die Insert-Operation alles Andere als arbeitsintensiv und verläuft in konstant vielen Schritten: Wir erzeugen einen neuen Baum, der nur aus einem Knoten mit dem in den Heap einzufügenden Schlüssel besteht und aktualisieren den min-Zeiger.

Die bisher betrachteten Operationen zerstückeln den Heap und die DeleteMin-Operation trägt die Hauptlast, nämlich die Vereinigung der Teilbäume.

Algorithmus 9.13 **Die DeleteMin-Operation für Fibonacci-Heaps**

(1) Entferne das Minimum.

Kommentar: Die Fibonacci-Eigenschaft garantiert, dass ein Knoten vom Grad k mindestens F_{k+2} Nachfahren besitzt. Andererseits gilt $F_{k+2} \geq \phi^k$ nach (9.7). Der Baum des Minimums wird also in höchstens $k \leq \log_\phi n$ neue Bäume aufgespalten, denn es gilt $\phi^k \leq n$ für die Knotenzahl n des Heaps.

(2) Der „Aufräum“-Schritt:

WHILE (es gibt Bäume T_1 und T_2 mit Wurzeln gleichen Grades) DO

Sei T_1 der Baum mit dem größeren Schlüssel. Hänge den Baum T_1 unter die Wurzel von T_2.

(3) Aktualisiere den min-Zeiger.

Lemma 9.14 *Nach jeder Operation (Insert, DeleteMin oder DecreaseKey) besteht die Datenstruktur weiterhin aus Fibonacci-Bäumen, d.h. jeder Knoten vom Grad k hat mindestens F_{k+2} Nachfahren.*

Beweis: Wir behandeln die einzelnen Operationen getrennt. Die Insert Operation generiert einen Baum, der nur aus einem Knoten besteht. Wegen $F_{0+2} = 1$ ist dies ein Fibonacci-Baum.

Die DeleteMin Operation entfernt die Wurzel eines Fibonacci-Baums und erzeugt neue Bäume. Als Unterbäume eines Fibonacci-Baums erfüllen diese neuen Bäume ebenfalls die Fibonacci-Eigenschaft. Sei k der Wurzelgrad der beiden Bäume T_1 und T_2, die wir in einem „Aufräum“-Schritt vereinigen. Die neue Wurzel hat den Grad $k+1$. Da T_1 und T_2 Fibonacci-Bäume sind, hat die neue Wurzel mindestens

$$F_{k+2} + F_{k+2} \geq F_{k+1} + F_{k+2} = F_{(k+1)+2}$$

viele Nachfahren, so dass der neue Baum die Fibonacci-Eigenschaft erfüllt.

Als letzte behandeln wir die DecreaseKey Operation und nehmen an, dass eine Anwendung von DecreaseKey den Baum T erzeugt. Wir zeigen durch Induktion über k, dass jeder Knoten v von T vom Grad k mindestens F_{k+2} Nachfahren hat.

Wenn v keine Kinder hat ($k = 0$), dann hat v nur sich selbst als Nachfahren. Die Behauptung für $k = 0$ folgt deshalb aus $F_2 = 1$. Wenn v genau ein Kind hat ($k = 1$), dann hat v mindestens sich selbst und das Kind als Nachfahren. Die Behauptung für $k = 1$ folgt also aus $F_3 = F_2 + F_1 = 1 + 1 = 2$.

Induktionsschluss : Der Knoten v habe $y_1, \ldots, y_{k+1}$ als Kinder, wobei y_i das i-älteste Kind sei. Die Prozedur „Aufräumen“ hat zu irgendeinem Zeitpunkt y_i zu einem Kind von v gemacht. Zu diesem Zeitpunkt waren $y_{i+1}, y_{i+2}, \ldots, y_{k+1}$ keine Kinder von v. Damit hatte y_i zum Zeitpunkt der Vereinigung den gleichen Grad wie v, also mindestens $i - 1$. Da y_i in der Zwischenzeit höchstens ein Kind verloren hat, ist sein aktueller Grad mindestens $i - 2$. Nach Induktionsannahme hat y_i mindestens F_i Nachfahren. Damit hat v wegen Identität (9.7) mindestens

$$1 + F_1 + F_2 + \cdots + F_{k+1} = F_{k+3}$$

Nachfahren. □

Schritt (1) der DeleteMin Operation läuft in Zeit $O(1)$, während Schritt (2) Zeit proportional zur Anzahl der Bäume benötigt. Schritt (3), die Aktualisierung des Minimum-Zeigers, gelingt schließlich in Zeit $O(\log_2 n)$, da nur die Grade 0 bis $\log_\phi n$ auftreten können.

Wir möchten zeigen, dass die DeleteMin Operation die amortisierte Laufzeit $O(\log_2 n)$ besitzt. Deshalb ordnen wir die Kosten für die Schritte (1) und (3) der DeleteMin Operation, die Kosten für Schritt (2) aber denjenigen Operationen zu, die die Bäume verursachten.

Der Insert-Operation ordnen wir eine Kosteneinheit für die Generierung eines neuen Baums und eine Kosteneinheit für den wirklich ausgeführten Schritt zu und wir erhalten konstante amortisierte Laufzeit.

Die DecreaseKey-Operation trennt möglicherweise einen Knoten ab und trägt dafür vier Kosteneinheiten. Eine Kosteneinheit wird für die Abtrennung bezahlt, zwei Kosteneinheiten werden (als „Ablöse“) dem Vater und eine Kosteneinheit wird der DeleteMin-Operation gutgeschrieben, da ein neuer Baum erzeugt wurde. Möglicherweise wird auch der Vater abgehängt. Diese Abhänge-Operation kostet ebenfalls vier Kosteneinheiten, die der Vater mit der Ablöse der beiden abgetrennten Kinder begleicht. Die amortisierte Laufzeit der DecreaseKey Operation ist also höchstens vier.

Wir fassen unsere Resultate über die amortisierte Laufzeit der betrachten Operationen zusammen.

Satz 9.15 *Man kann auf einem anfänglich leeren Fibonacci-Heap die Operationen*

(a) Insert in amortisierter Zeit $O(1)$,

(b) DecreaseKey in amortisierter Zeit $O(1)$ und

(b) DeleteMin in amortisierter Zeit $O(\log_2 n)$

ausführen, wobei n die Anzahl der Insert-Operationen sei.

Neben den hier besprochenden Operationen unterstützen Fibonacci-Heaps auch folgende Operationen [CLR]:

- Vereinigung zweier Fibonacci-Heaps in amortisierter Zeit $O(1)$ und
- Delete in amortisierter Zeit $O(\log_2 n)$.

9.4 Suche in Listen

Wir beschreiben zuerst on-line Algorithmen für das eingeschränkte **List-Update** Problem:

> Ein on-line Algorithmus A beginnt mit einer nicht-sortierten Liste L von Daten. Eine Folge $\sigma = (\sigma_1, \ldots, \sigma_m)$ von Lookup-Anfragen ist zu beantworten, wobei für jede Anfrage σ_i die gegenwärtige Liste von ihrem Anfang her zu durchlaufen ist bis das gesuchte Element gefunden ist. Das gesuchte Element darf dann kostenfrei an eine beliebige Stelle näher am Listenanfang gebracht werden. Desweiteren dürfen benachbarte Listenelemente an jeder Stelle der Liste nach Beantwortung der Anfrage σ_i vertauscht werden: jede solche Vertauschung ist allerdings *kostenpflichtig*.
>
> Wenn sich σ_i in Phase i an Stelle l_i der Liste befindet und wenn während der Beantwortung von σ_i genau v_i kostenpflichtige Vertauschungen durchgeführt werden, dann verursacht Algorithmus A die Kosten
>
> $$\text{Kosten}_A(L, \sigma) = \sum_{i=1}^{m} (l_i + v_i).$$

Die Move-to-front Regel bewegt das nachgefragte Listenelement an den Anfang der Liste, kostenpflichtige Vertauschungen hingegen werden nicht ausgeführt. Um Move-to-front zu evaluieren, vergleichen wir die Strategie für jede Liste L und für jede Folge σ von Zugriffen mit einem optimalen off-line Algorithmus Opt. Man beachte, dass Opt die Liste L wie auch die Folge σ von vornherein kennt, (kostenpflichtige) Vertauschungen an beliebiger Stelle ausführen darf und damit einen anscheinend unfairen Vorteil hat. Umso überraschender, dass die Move-to-front Strategie fast mithalten kann, denn sie besitzt den optimalen *Wettbewerbsfaktor* 2, d.h. es gilt

$$\text{Kosten}_{\text{Move-to-front}}(L, \sigma) \leq 2 \cdot \text{Kosten}_{\text{Opt}}(L, \sigma)$$

für jede Liste L und jede Folge σ. Weiterhin zeigen wir, dass jeder andere on-line Algorithmus mindestens den Wettbewerbsfaktor 2 hat.

Satz 9.16

(a) Jede deterministische on-line Strategie besitzt einen Wettbewerbsfaktor $c \geq 2$.

(b) Move-to-front besitzt den optimalen Wettbewerbsfaktor 2.

Beweis (a): Sei A ein beliebiger deterministischer on-line Algorithmus. Wir nehmen an, dass die Liste n Elemente speichert. Unsere Anfragefolge wird m-mal das jeweils an letzter Position stehende Listenelement nachfragen und A wird bei m Anfragen die Kosten $m \cdot n$ verursachen.

Wir lassen A gegen einen off-line Algorithmus antreten, der zuerst die jeweiligen Häufigkeiten bestimmt, mit der ein Listenelement nachgefragt wird. In einem zweiten Schritt wird die Liste umorganisiert, so dass die Liste gemäß fallenden Häufigkeiten sortiert ist. Offensichtlich fallen hierzu nur die Kosten der $\frac{n \cdot (n-1)}{2}$ Vertauschungsoperationen an.

Danach werden, ohne irgendein weiteres Listenelement zu verschieben, alle Anfragen durch einen entsprechenden Listendurchlauf beantwortet. Die hierbei anfallenden Kosten sind durch $m \cdot \frac{n+1}{2}$ beschränkt und insgesamt ergeben sich für hinreichend großes m Kosten von

$$m \cdot \frac{n+1}{2} + \frac{n \cdot (n-1)}{2} \approx m \cdot \frac{n+1}{2}$$

und die Behauptung folgt.

(b) Sei $\sigma = (\sigma_1, \ldots, \sigma_m)$ eine Anfragefolge der Länge m. Wir benutzen eine Potentialfunktion Φ, um Move-to-front mit einer optimalen off-line Strategie OPT zu vergleichen.

Wir sagen, dass das Paar (x, y) zum Zeitpunkt t, also vor Beantwortung der Nachfrage nach σ_{t+1}, eine Inversion ist, wenn x zum Zeitpunkt t vor y in der Move-to-front Liste erscheint, während x zum selben Zeitpunkt nach y in der Liste von OPT auftaucht. Wir setzen

$$\Phi_t = \text{ Anzahl der Inversionen zum Zeitpunkt } t$$

und beachten, dass $\Phi_0 = 0$, denn beide Listen sind zu Anfang identisch. Weiterhin seien $\text{Kosten}_t(\text{MTF})$ und $\text{Kosten}_t(\text{OPT})$ die Kosten von Move-to-front und OPT bei der Beantwortung der Anfrage σ_t. Wir erinnern daran, dass $\text{Kosten}_t(\text{MTF}) + \Phi_t - \Phi_{t-1}$ die amortisierten Kosten von Move-to-front bei der Beantwortung der Anfrage σ_t sind. Es genügt der Nachweis von

$$\text{Kosten}_t(\text{MTF}) + \Phi_t - \Phi_{t-1} \leq 2 \cdot \text{Kosten}_t(\text{OPT}) - 1, \tag{9.8}$$

denn nach Summation über alle Zeitpunkte t werden dann die amortisierten Gesamtkosten von Move-to-front durch die doppelten Gesamtkosten von OPT beschränkt.

Wir zeigen (9.8) für ein beliebiges t und nehmen an, dass $\sigma_t = x$. Sei K die Anzahl der Elemente, die sowohl in der Move-to-front Liste wie auch in der OPT-Liste vor Element x erscheinen und sei L die Anzahl der Elemente, die in der Move-to-front Liste vor x, aber in der OPT-Liste nach x erscheinen. Offensichtlich ist $\text{Kosten}_t(\text{MTF}) = K+L+1$ und $\text{Kosten}_t(\text{OPT}) \geq K + v_t + 1$, wenn Opt genau v_t kostenpflichtige Vertauschungen zum Zeitpunkt t ausführt.

Nach Beantwortung von σ_t wird x durch Move-to-front an den Listenanfang gebracht: L Inversionen werden beseitigt und K Inversionen werden neu geschaffen. Dann gilt

$$\begin{aligned} \text{Kosten}_t(\text{MTF}) + \Phi_t - \Phi_{t-1} &\leq (K + L + 1) + K - L + v_t \\ &= 2 \cdot K + v_t + 1 \\ &\leq \text{Kosten}_t(\text{OPT}) + K \\ &\leq 2 \cdot \text{Kosten}_t(\text{OPT}) - 1 \end{aligned}$$

und dies war zu zeigen. □

Aufgabe 84

In der *Transpose* Regel wird das gefundene Element um eine Position nach vorne gerückt, während die Listenanordnung in der *Lazy* Regel nicht verändert wird.

a) Beschreibe eine Folge von Zugriffen auf eine Liste $(x_1, \ldots, x_n)$, für die bei Anwendung der Move-to-Front Regel geringere Kosten entstehen als bei Lazy, und eine Folge von Zugriffen, für die bei Anwendung der Lazy Regel geringere Kosten entstehen als bei Move-to-front. Dabei soll der Unterschied jeweils möglichst groß sein.

b) Konstruiere eine Zugriffsfolge, für die die Transpose Regel bei einer Liste der Länge n um einen Faktor $\Omega(n)$ höhere Kosten verursacht als die Move-to-Front Regel.

Randomisierte Strategien erreichen bessere Wettbewerbsfaktoren:

Aufgabe 85

Betrachte nun folgende randomisierte Strategie *RMF* zur Selbstanordnung von Listen: Jedes Element x in der Liste ist mit einem Bit $b(x)$ markiert, welches bei einem Zugriff gekippt wird. Wenn das Bit von 0 auf 1 wechselt, wird das Element an den Anfang der Liste gebracht, sonst nicht. Zu Beginn werden alle Bits zufällig ausgewürfelt.

Zeige, dass die beschriebene Strategie höchstens Kosten $\frac{7}{4} \cdot \text{opt} - O(m)$ verursacht bei optimalen off-line-Kosten opt.

Hinweis: Gehe wie im Beweis von Satz 9.16 vor. Eine Inversion (y, x) (x kommt in der Liste einer optimalen Strategie A vor y, in der Liste von RMF hinter y vor) ist eine Typ-1-Inversion, wenn $b(x) = 0$, sonst eine Typ-2-Inversion. Sei ϕ_1 die Anzahl der Typ-1-Inversionen, ϕ_2 die Anzahl der Typ-2-Inversionen. Verwende $2\phi_2 + \phi_1$ als Potentialfunktion. Beachte, dass $b(y) = 1$ (für ein Listenelement y) mit Wahrscheinlichkeit $\frac{1}{2}$.

Wie nahe randomisierte Strategien an die Leistung des besten off-line Algorithmus herankommen, ist nicht bekannt:

Offenes Problem 5
Bestimme den optimalen Wettbewerbsfaktor randomisierter Strategien für das List-Update Problem. Der gegenwärtig beste Wettbewerbsfaktor ist 1.6 [ASW]. Weiterhin ist bekannt, dass der Wettbewerbsfaktor mindestens 1.5 beträgt [Te].

Literaturverzeichnis

[AKS] M. Agrawal, N. Kayal und N. Saxena, Primes is in P, `http://www.cse.iitk.ac.in/users/manindra/index.html`

[ASW] S. Albers, B. von Stengel und R. Werchner, A combined BIT and TIMESTAMP algorithm for the list update problem, *Information Processing Letters* (56), pp. 135-139, 1995.

[BF] I. Bárány und Z. Füredi, Computing the volume is difficult, *Proceedings of the 18.th Annual ACM Symposium of the Theory of Computing*, pp. 442-447, 1986.

[BE] A. Borodin und R. El-Yaniv, Online Computation and Competitive Analysis, *Cambridge University Press* 1998.

[B] J. Boyar, Infering sequences produced by pseudo-random number generators, *J. of the Assoc. Comput. Mach.* 36, pp. 129-141, 1989.

[CLR] T. Cormen, C.E. Leiserson und R.L. Rivest, Introduction to Algorithms, *MIT Press und McGraw Hill*, 1990.

[C] T.M. Cover, Universal Portfolios, *Math. Finance* 1(1), pp. 1-29, 1991.

[DEKH] R. Durbin, S. Eddy, A. Krogh und G. Mitchison, Biological Sequence Analysis, *Cambridge University Press*, 1998.

[F] W. Feller, An Introduction to Probability Theory and its Applications, vol 1, 3rd edition, John Wiley & Sons, 1968.

[G] E.F. Grove, The harmonic online k-server algorithm is competitive, *Proc. of the 23rd Annual ACM Symposium on Theory of Computing*, pp. 260-266, 1991.

[HILL] J. Hastad, R. Impagliazzo, L.A. Levin und M. Luby, Construction of pseudo-random generators from any one-way function, *SIAM Journal on Computing*, 1999.

[HSSW] T. Hofmeister, U. Schöning, R. Schuler und O. Watanabe, A Probabilistic 3-SAT Algorithm Further Improved. *Proceedings of the Symposium on Theoretical aspects of Computer Science*, pp. 192-202, 2002.

[IR] S. Irani und R. Rubinfeld, A Competitive 2-Server Algorithm , *Information Processing Letters (39)*, pp. 85-91, 1991.

[IW97] R. Impagliazzo und A. Wigderson, $P = BPP$, unless E has subexponential circuits: Derandomizing the XOR-lemma, *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pp. 220-229, 1997.

[JAGS] D.S. Johnson, C.R. Aragon, L.A. McGeoch und C. Schevon, Simulated Annealing: An experimental Evaluation, Part I: Graph Partioning, *Operation Research (37)*, pp. 865-892, 1989.

[KKT] C. Kaklamanis, D. Krizanc und T. Tsantilas, Tight bounds for oblivious routing in the hypercube, *Proceedings of the 3rd Symposium on Parallel Algorithms und Architectures*, pp. 31-36, 1991.

[KLL84] R.A. Kannan, A. Lenstra und L. Lovasz, Polynomial factorization and nonrandomness of bits of algebraic and some transcendental numbers, *Proceedings 16th Annual ACM Symposium on Theory of Computing*, pp. 191-200, 1984.

[KLS] R. Kannan, L. Lovász und M. Simonovits, Random Walks and an $O(n^5)$ Volume Algorithm, Preprint, 1996.

[K] V. King, A simpler minimum spanning tree verification algorithm, *Algorithmica* 18, pp. 263-270, 1997.

[Ko] E. Koutsoupias, Weak adversaries for the k server problem, *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pp. 444-449, 1999.

[LS] L. Lovász und M. Simonovits, The mixing Rate of Markoff Chains, an isoperimetric Inequality and Computing the Volume, *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pp. 346-254, 1990.

[MP] M. Minsky und S. Papert, Perceptrons: An Introduction to Computational Geometry, 3rd edition, *MIT Press*, 1988.

[ML] M. Mitzenmacher and E. Upfal, Randomized algorithms and probabilistic analysis, Cambridge University Press, 2005.

[MR] R. Motwani und P. Raghavan, Randomized Algorithms, *Cambridge University Press*, 1995.

[R] F. Rosenblatt, The Perceptron: A probabilistic Model for Information Storage and Organization in the Brain, *Psychological Review*, 65, pp. 386-407, 1958.

[SH] G.H. Sasaki und B. Hajek, The Time Complexity of Maximum Matching by simulated Annealing, *Journal of the ACM (35)*, pp. 387-403, 1988.

[ST] D.D. Sleator und R.E. Tarjan, Self-adjusting Binary Trees, *Journal of the ACM* (32), pp. 652-682, 1985.

[SW] M. Saks und A. Wigderson, Probabilistic boolean decision trees and the complexity of evaluating game trees, *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pp. 29-38, 1986.

[Ta] M. Tarsi, Optimal search on some game trees, *Journal of the ACM* (30), pp. 389-396, 1983.

[Te] B. Teia, A lower bound for randomized list update algorithms, *Information Processing Letters* (47), pp. 5-9, 1993.

[V] B. Vöcking, How asymmetry helps load balancing, *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pp. 131-140, 1999.

[Vaz] V.V. Vazirani, Approximation Algorithms, *Springer Verlag* 2001.