Platzeffiziente Hashverfahren mit garantierter konstanter Zugriffszeit

Dissertation zur Erlangung des akademischen Grades
Doctor rerum naturalium
(Dr. rer. nat.)

vorgelegt der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von

Dipl.-Inf. Christoph Weidling

Gutachter:

1. Univ.-Prof. Dr. rer. nat. (USA) M. Dietzfelbinger,
   Technische Universität Ilmenau
2. Univ.-Prof. Dr. rer. nat. A. Goerdt,
   Technische Universität Chemnitz
3. Univ.-Prof. Dr. rer. nat. P. Sanders,
   Universität Kaiserslautern

Vorgelegt am: 7. Juli 2004

Verteidigt am: 4. Oktober 2004

urn:nbn:de:gbv:ilm1-2004000134

Zusammenfassung

Wir stellen ein neues Verfahren zur Konstruktion einer minimalen perfekten Hashfunktion (MPHF) und ein neues cachefreundliches dynamisches Wörterbuch vor, beschreiben die neuen Verfahren algorithmisch und analysieren sie hinsichtlich Platzbedarf und Laufzeit. Für die Analyse nehmen wir an, dass die in den Verfahren benutzten Hashfunktionen volle Unabhängigkeit gewährleisten. Schließlich werden wir jeweils experimentelle Resultate angeben und interpretieren.

Wir zeigen, dass man eine minimale perfekte Hashfunktion für n Schlüssel mit einem Platzbedarf von $(0.93+\varepsilon)n$ Wörtern in erwarteter Zeit $O(n)$ realisieren kann. Zur Auswertung der MPHF werden 2 Hashfunktionen berechnet und zwei Speicherzugriffe durchgeführt.

Unser dynamisches Wörterbuch benötigt $(1+\varepsilon)n$ Platz für ein beliebiges $\varepsilon > 0$. Bei der Suche müssen 2 Hashfunktionen ausgewertet und $2d$ Speicherzellen inspiziert werden, die in zwei zusammenhängenden Speicherbereichen liegen, wobei $d \geq 1 + 3.26 \cdot \ln(1/\varepsilon)$. Wir können zeigen, dass für $d \geq 90.1 \cdot \ln(1/\varepsilon)$ die erwartete Einfügezeit für einen neuen Schlüssel konstant ist. Am Ende werden wir zeigen, wie man dynamisches Hashing mit Zeichenketten cachefreundlich realisieren kann.

Abstract

We present a new algorithm for a minimal perfect hash function (MPHF) and a new dynamic cache friendly dictionary. We describe the procedures and analyse them with respect to space and time. For our analysis we assume full randomness of hash functions which are used by the algorithms. Finally we give experimental results and discuss them.

We show that it is possible to construct a minimal perfect hash function which stores n keys and consumes $(0.93+\varepsilon)n$ words. It takes $O(n)$ expected time to build such a MPHF. To evaluate our MPHF only 2 memory cells have to be read and only 2 hash functions have to be evaluated.

Our dynamic dictionary consumes $(1+\varepsilon)n$ space for an arbitrary $\varepsilon > 0$. For a `lookup` operation only 2 hash functions have to be evaluated and $2d$ memory cells have to be inspected, which are located in 2 contiguous blocks for each $d \geq 1 + 3.26 \cdot \ln(1/\varepsilon)$. Further we proof that for each $d \geq 90.1 \cdot \ln(1/\varepsilon)$ the expected time for inserting a new key is constant. Finally we show how to adapt our dictionary to use strings as keys in a cache friendly manner.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die vorliegende Arbeit stellt die Ergebnisse unserer Untersuchungen über Wörterbuch-Datenstrukturen auf Basis von Hashing dar.

Wir beschäftigen uns mit Wörterbüchern, die eine konstante Zugriffszeit im schlechtesten Fall (*worst case*) haben. Dabei werden wir ein neues Verfahren für ein statisches (Kapitel 2) und ein neues cachefreundliches dynamisches Wörterbuch (Kapitel 3) vorstellen. Nach einem kurzen Überblick über den aktuellen Stand der Forschung werden wir jeweils die neuen Verfahren beschreiben und hinsichtlich Platzbedarf und Laufzeit analysieren. Schließlich werden wir jeweils experimentelle Resultate angeben und interpretieren. Unser dynamisches Wörterbuch werden wir am Ende so modifizieren, dass es sich auch für Zeichenketten eignet.

Die hier gezeigten Ergebnisse entstanden während der letzten vier Jahre in Kooperation mit M. Dietzfelbinger, bei dem ich mich ganz herzlich an dieser Stelle für die inspirierende und fruchtbringende Zusammenarbeit und Betreuung bedanken möchte.

1.1 Wörterbücher

Ein *Wörterbuch* W ist eine Datenstruktur, in der eine Menge von Datensätzen gespeichert ist. Jeder Datensatz ist durch einen *Schlüssel* gekennzeichnet, über den der Datensatz eindeutig identifiziert werden kann.

Die Schlüssel sind Elemente eines *Universums* U. Typische Beispiele für U sind

- die Menge aller 32-Bit-Ganzzahlen,
- die Menge aller Zeichenketten, die sich aus ASCII-Zeichen zusammensetzen lassen oder

- die Menge aller Personen eines Landes gekennzeichnet durch Name, Vorname und Geburtstag. An diesem Beispiel sieht man, dass man bei der praktischen Anwendung von Wörterbüchern Probleme bekommen kann, weil in diesem Beispiel nicht gewährleistet ist, dass der Schlüssel eindeutig ist.

Wir unterscheiden zwischen *dynamischen* und *statischen* Wörterbüchern. Zu einem statischen Wörterbuch gehören die Operationen `lookup`(s) und `build`(S). Dynamische Wörterbücher bieten die Operationen `insert`(s,w), `lookup`(s) und `delete`(s). Diese Operationen arbeiten auf einem Wörterbuch W wie folgt:

- `lookup`(s) sucht den Datensatz mit dem Schlüssel s in W und gibt den zu s gehörenden Datensatz zurück, sofern s in W vorhanden ist. Ist kein solcher Datensatz in W vorhanden, wird eine Fehlermeldung ausgegeben.

- `insert`(s,w) fügt den Datensatz w mit dem Schlüssel s in W ein, falls noch kein Datensatz mit dem Schlüssel s in W vorhanden ist. Falls bereits ein solcher Datensatz vorhanden ist, wird eine Fehlermeldung ausgegeben.

- `delete`(s) löscht den Datensatz mit dem Schlüssel s aus W. Ist kein Datensatz mit dem Schlüssel s in W vorhanden, wird eine Fehlermeldung ausgegeben.

- `build`(S) fügt alle Schlüssel $x \in S$ in W ein.

Anwendungen für statische Wörterbücher sind beispielsweise Nachschlagewerke auf CD-ROM, wie etwa ein Telefonbuch. Dabei ist ein Eintrag durch Namen und Wohnort gekennzeichnet und der Datensatz ist die Telefonnummer. Dynamische Wörterbücher findet man

- in Rechtschreibehilfen von Textprogrammen. Diese Rechtschreibehilfen kann man mit unbekannten Begriffen erweitern.

- in relationalen Datenbanken. Indizes über Schlüssel einer Tabelle sind nichts anderes als Wörterbücher.

- in Nameservern (*Domain Name Service, DNS)* des Internet. Der Schlüssel ist der Name eines Rechners, etwa `www.tu-ilmenau.de`, und der zugehörige Datensatz ist die IP-Adresse des Rechners.

- in Netzwerkroutern. Hier ist es besonders wichtig, dass Pakete sehr schnell geroutet werden können (Echtzeitanforderungen).

Im Folgenden wollen wir uns darauf beschränken, ausschließlich die Schlüssel in einem Wörterbuch zu speichern. Denn wenn wir wissen, wie wir Schlüssel einfügen, suchen und löschen können, ist es leicht, mit den Schlüsseln auch Datensätze zu speichern.

1.2 Hashfunktionen

Eine Hashfunktion h berechnet aus einem Schlüssel $s \in U$ eine natürliche Zahl. Wir wollen den Bildbereich der Hashfunktion mit $\{0, \ldots, m-1\}$ für eine natürliche Zahl m festlegen. Zur Abkürzung schreiben wir anstelle von $\{0, \ldots, m-1\}$ einfach $[m]$. Formal schreiben wir also für eine Hashfunktion $h\colon U \to [m]$.

Hashfunktionen werden gern benutzt, um für einen Schlüssel seine Adresse im Speicher eines Computers zu berechnen. Von einer Hashfunktion h werden in verschiedenen Anwendungen „gute“ Eigenschaften verlangt, wie zum Beispiel die folgenden:

1. Für zwei verschiedene Schlüssel $x, y \in S$ soll nach Möglichkeit $h(x) \neq h(y)$ gelten. Gilt $h(x) = h(y)$, sprechen wir von einer *Kollision* der Schlüssel x und y unter h.
2. $h(x)$ soll sich für alle $x \in U$ schnell[1] berechnen lassen.
3. Die Beschreibung von h soll wenig Platz benutzen.

1.3 Universelle Hashklassen, volle Unabhängigkeit und uniformes Hashing

Die Idee hinter dem Konzept des Universellen Hashings ist, für jede gegebene Schlüsselmenge S die Hashfunktion h aus einer Klasse $\mathcal{H}$ von Hashfunktionen zufällig zu wählen. Dieses Zufallsexperiment induziert einen Wahrscheinlichkeitsraum.

Definition 1. Eine Klasse $\mathcal{H} \subseteq \{h \mid h\colon U \to [m]\}$ von Hashfunktionen heißt c-universell, falls für alle $x, y \in U$ mit $x \neq y$ gilt:

$$\forall h \in \mathcal{H}\colon \mathbf{Prob}(h(x) = h(y)) \leq \frac{c}{m}$$

[1] Was „schnell“ heißt, hängt vom Kostenmaß ab. Das klassische Kostenmaß ist die Anzahl der Operationen, die benötigt werden, um h auszuwerten. Für die Geschwindigkeit der Auswertung von h auf aktuellen Rechnern tragen auch andere Kosten bei, wie die Anzahl der Cache-Fehler oder der Grad der Parallelisierbarkeit.

Die Klasse heißt k-fach unabhängig, wenn für k verschiedene Schlüssel $x_1, \ldots, x_k \in U$ für eine zufällig gewählte Funktion h die Werte $h(x_1), \ldots, h(x_k)$ uniform und unabhängig über $[m]$ verteilt sind.

Dieses Konzept wurde von Carter und Wegman eingeführt [CW79, WC81]. Man weiß zum Beispiel, dass die Menge aller Polynome vom Grad kleiner als k über einem passend gewählten Körper, wie zum Beispiel $\mathbb{Z}_p$ für eine Primzahl p, k-fache Unabhängigkeit gewährleistet.

Wenn die $h(x)$ für $x \in U$ uniform und unabhängig verteilt sind, so spricht man von uniformem Hashing. Dies ist ein Idealzustand für Hashfunktionen, den man gern erreichen möchte. In jüngster Vergangenheit wurden große Anstrengungen dahingehend unternommen, eine billige Möglichkeit zu finden, uniformes Hashing zu simulieren.

Siegel [Sie89] stellt eine Klasse von Hashfunktionen $h\colon [n^k] \to [n]$ vor, die n^μ-fach unabhängig ist für ein $0 < \mu < 1/k$. Eine solche Funktion belegt n^ζ Platz für ein $\zeta < 1$. Allerdings ist die Auswertung einer solchen Funktion sehr aufwändig. Wenn U gegenüber $[m]$ sehr groß ist, kann der Exponent μ nicht groß werden.

Östlin und Pagh [ÖP03] zeigen, wie man eine Funktion $h\colon U \to [m]$ wählen kann, die $8n + O(n^\zeta)$ Wörter aus $[m]$ und $O(n^\zeta)$ Wörter aus $[n]$ benutzt und für jede Menge $S \subseteq U$ mit $|S| = n$, mit Wahrscheinlichkeit $O(n^{-c})$ für ein beliebiges c die Werte $h(x)$, $x \in S$, uniform und unabhängig über $[m]$ verteilt. Während der Auswertung von h müssen 3 Siegel-Funktionen ausgewertet werden.

Dietzfelbinger und Woelfel [DW03] zeigen eine Alternative, wie man eine solche Funktion konstruieren kann, die mit $2(1+\varepsilon)n$ Wörtern aus U und $(1+\varepsilon)n$ Wörtern aus $[m]$ auskommt. Jedoch wird dabei während der Auswertung von h auf die Auswertung von Siegel-Funktionen verzichtet; man kommt mit lediglich einigen Additionen und Divisionen aus.

Um den Platzbedarf zu senken, kann man sich des folgenden Tricks bedienen: Man teilt die Schlüsselmenge S in $\ln n$ disjunkte Mengen der Größe höchstens $\frac{1+\varepsilon/2n}{\ln n}$ auf. Eine solche Aufteilung kann beispielsweise durch ein Polynom festen Grades erreicht werden. Innerhalb einer solchen Menge verwendet man eine der Konstruktionen, die linearen Platz benötigen. Dabei stellt man fest, dass für alle Mengen die gleichen Hashfunktionen verwendet werden können, so dass man mit insgesamt sublinearem Platz auskommt [FPSS03].

Bemerkung 1. In der vorliegenden Arbeit gehen wir davon aus, dass die in den Verfahren benutzten Hashfunktionen volle Zufälligkeit gewährleisten.

1.4 Einige Ungleichungen

Wir wollen hier einige Ungleichungen angeben, die in der Arbeit benutzt werden. Wir geben keine Beweise für diese Beziehungen an, da diese schon hinreichend untersucht sind.

Wohlbekannt ist die folgende Ungleichung, die für alle $x \in \mathbb{R}$ gilt:

$$1 + x \leq e^x \tag{1.1}$$

Eine Möglichkeit, um $n!$ abzuschätzen, ist die Stirlingsche Formel. Wir benutzen hier die Variante aus [CLRS01, Abschnitt 3.2]:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{1.2}$$

mit $\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}$. Damit folgt

$$n! < \sqrt{2\pi n} \left(\frac{n}{e}\right)^n. \tag{1.3}$$

Folgende Abschätzung für $n!$ ist wohlbekannt und ohne die Stirlingsche Formel beweisbar (zum Beispiel [DK03]):

$$e \left(\frac{n}{e}\right)^n < n! < en \left(\frac{n}{e}\right)^n \tag{1.4}$$

Wir benutzen ständig Binomialkoeffizienten. Diese können mit der Stirlingschen Formel abgeschätzt werden. Wir wählen jedoch häufig eine andere Abschätzung, die etwas bequemer ist (zum Beispiel [FPSS03]):

$$\binom{n}{k} \leq \frac{n^n}{k^k (n-k)^{n-k}} = \left(\frac{1}{\mu^\mu (1-\mu)^{(1-\mu)}}\right)^n \tag{1.5}$$

mit $\mu = k/n$. Eine weitere bequeme, obgleich schwächere, Abschätzung ist die folgende:

$$\binom{n}{k} \leq \left(\frac{en}{k}\right)^k \tag{1.6}$$

Weiterhin sei noch die Tschebyschewsche[2] Ungleichung erwähnt: Sei Z eine Zufallsgröße, dann gilt

$$\mathbf{Prob}(|Z - \mathbf{E}(Z)| \geq \delta) \leq \frac{\mathbf{Var}(Z)}{\delta^2}. \tag{1.7}$$

[2] Чебышев Пафнутий Львович

Ganz wichtig ist für uns ist eine Chernoff-Hoeffding-Schranke [HR90]. Sind $X_1, \ldots, X_n \in \{0,1\}$ unabhängige Zufallsvariablen und $X = \sum_{i=1}^{n} X_i$, so gilt für $a \geq 0$:

$$\mathbf{Prob}(X - \mathbf{E}(X) \geq a) \leq \left(\frac{\mathbf{E}(X)}{\mathbf{E}(X)+a}\right)^{\mathbf{E}(X)+a} \left(\frac{n-\mathbf{E}(X)}{n-\mathbf{E}(X)-a}\right)^{n-\mathbf{E}(X)-a} \tag{1.8}$$

Wir verwenden die Ungleichung in der Form

$$\mathbf{Prob}(X \geq T) \leq \left(\frac{\mathbf{E}(X)}{T}\right)^{T} \left(\frac{n-\mathbf{E}(X)}{n-T}\right)^{n-T} \tag{1.9}$$

für $T \geq E(X)$.

1.5 Modelle von Zufallsgraphen

In dieser Arbeit benutzen wir insbesondere im Kapitel 2 zwei Modelle von Zufallsgraphen [Bol85, Kapitel II]:

Das Modell $\mathfrak{G}_{n,p}$: In diesem Modell werden alle Graphen mit n Knoten betrachtet, in denen alle Kanten unabhängig mit einer Wahrscheinlichkeit p gewählt werden. Wir betrachten in dieser Arbeit dünne Graphen mit $p = p(n) = d/n$. Die erwartete Zahl von Kanten in einem solchen Graphen ist $\binom{n}{2}p = d\frac{n-1}{2} \approx dn/2$.

Das Modell $\mathfrak{G}_{n,m}$: Hier werden alle Graphen mit n Knoten und m Kanten betrachtet, wobei jeder Graph mit gleicher Wahrscheinlichkeit gewählt werden kann. Weil es $\binom{N}{m}$, $N = \binom{n}{2}$, Graphen mit n Knoten und m Kanten gibt, kann jeder Graph mit einer Wahrscheinlichkeit von $1/\binom{N}{m}$ gewählt werden.

Unter bestimmten Voraussetzungen lassen sich Aussagen, die man für das $\mathfrak{G}_{n,m}$-Modell trifft, auf das entsprechende $\mathfrak{G}_{n,m}$-Modell übertragen und umgekehrt [Bol85, Abschnitt II.1]. In solchen Fällen kann man sich das bequemere Modell aussuchen.

1.6 Zusammenfassung der Ergebnisse

In Kapitel 2 modifizieren wir ein Verfahren zur Konstruktion einer minimalen perfekten Hashfunktion (MPHF) aus [CHM92, CHM97] so, dass anstelle von $(2+\varepsilon)n$ Wörtern aus $[n]$ nur noch $(1.16+\varepsilon)n$ Wörter aus $[n]$ benötigt

werden. Durch Einführung eines sogenannten *Undo One*-Schrittes können wir zeigen, dass sogar $(0.93 + \varepsilon)n$ Wörter aus $[n]$ ausreichen. Dabei ist die Auswertung der so konstruierten Funktionen einfach: Es müssen lediglich 2 Hashfunktionen ausgewertet und 2 Zugriffe in ein Feld durchgeführt werden.

Die Analyse des Verfahrens hängt eng mit der Analyse des 2-Kerns von Zufallsgraphen zusammen. Der 2-Kern ist der größte Teilgraph von G, in dem alle Knoten einen Grad von mindestens 2 haben. Um die Analyse durchzuführen, bestimmen wir für $p = d/n$ die erwartete Anzahl von Kanten, die wenigstens einen Knoten außerhalb des 2-Kerns besitzen, für das $\mathfrak{G}_{n,p}$-Modell [Bol85] asymptotisch exakt.

Experimente lassen vermuten, dass es mit dem vorgestellten Verfahren möglich ist, mit nur etwa $0.35n$ Wörtern auszukommen.

In Kapitel 3 geben wir ein neues auf *Cuckoo Hashing* basierendes dynamisches Wörterbuch an, das n Schlüssel in $m = (1 + \varepsilon)n$ Zellen für ein beliebiges $\varepsilon > 0$ speichern kann. Dieses Kapitel stellt den Haupteil der vorliegenden Arbeit dar.

Das Wörterbuch ist so angelegt, dass bei der Suche nach einem Schlüssel lediglich zwei Hashfunktionen ausgewertet und $2d$ Speicherzellen inspiziert werden müssen, die in zwei zusammenhängenden Speicherbereichen liegen. Dabei ist d eine Konstante. Wir zeigen, dass für $d \geq 1 + 3.26 \cdot \ln(1/\varepsilon)$ mit hoher Wahrscheinlichkeit n Schlüssel in m Zellen gespeichert werden können.

Wir zeigen weiter, dass für $d \geq 90.1 \cdot \ln(1/\varepsilon)$ die erwartete Einfügezeit für einen neuen Schlüssel konstant ist.

Weil bei den Operationen, die auf dem Wörterbuch ausgeführt werden, stets Speicherbereiche in zusammenhängenden Blöcken der Größe d inspiziert werden, nennen wir das Wörterbuch *cachefreundlich*.

In Experimenten untersuchen wir die Zahl der Cache-Fehler für verschiedene d und ε und geben gemessene Laufzeiten für die verschiedenen Operationen an. Die Experimente legen nahe, dass das neue Wörterbuch eine Alternative zum d-ären Cuckoo Hashing [FPSS03] und zum Linearen Sondieren ist. Außerdem zeigt sich in den Experimenten, dass man d viel kleiner als hier angegeben wählen kann.

Schließlich untersuchen wir eine spezielle Variante des Wörterbuches, um Zeichenketten (*Strings*) laufzeiteffizient zu speichern. Zusätzlich zu den Zeigern auf die Positionen der Strings im Hauptspeicher werden im Wörterbuch Fingerabdrücke der Strings geseichert. Dadurch wird auf Kosten des Platzes die Zahl von String-Vergleichen reduziert.

Für eine Einordnung unserer erzielten Resultate in bisher bekannte Ergebnisse sei auf die Einleitungen der jeweiligen Kapitel verwiesen.

Kapitel 2

Ein Verfahren für eine minimale perfekte Hashfunktion

In diesem Kapitel stellen wir ein Verfahren zur Konstruktion einer minimalen perfekten Hashfunktion (MPHF) vor. Dazu betrachten wir eine gegebene Schlüsselmenge S mit n Schlüsseln aus einem Universum U.

Eine MPHF W ist nichts anderes als eine Funktion, welche die Schlüsselmenge S bijektiv auf die Menge $[n]$ abbildet. Sie heißt deswegen minimal, weil kein Wert aus dem Bildbereich $[n]$ „verschenkt“ wird.

Ein Hauptaugenmerk bei der Konstruktion einer MPHF W liegt auf dem von W benutzten Platz. Bei den Funktionen, die wir betrachten, sind das cn Wörter aus $[n]$ für eine Konstante $c > 0$. Dabei soll c so klein wie möglich sein.

Czech, Havas und Majewski stellen einen Algorithmus vor, der eine solche MPHF mit Hilfe von Zufallsgraphen erzeugt [CHM92, CHM97]: Für eine gegebene Schlüsselmenge S, $|S| = n$, werden so lange Hashfunktionen $h_0, h_1 : U \to [m]$ mit $m = \lfloor (2+\varepsilon)n \rfloor$ zufällig gewählt, bis der Graph mit der Knotenmenge $[m]$ und der Kantenmenge $\{\{h_0(x), h_1(x)\} \mid x \in S\}$ kreisfrei ist. Anschließend wird jeder Knoten v des Graphen mit einer Zahl $g[v]$ beschriftet, so dass die Funktion $W\colon x \mapsto (g[h_0(x)] + g[h_1(x)]) \bmod n$ eindeutig ist. Die in den genannten Arbeiten gegebene Analyse setzt volle Unabhängigkeit der Funktionen h_0 und h_1 voraus.

Der Faktor $2 + \varepsilon$ ist dabei nicht zu unterbieten. Das liegt daran, dass ein Zufallsgraph, der mehr als halb so viele Kanten wie Knoten besitzt, mit hoher Wahrscheinlichkeit einen Kreis enthält.

Für die Auswertung der so konstruierten MPHF reicht es, zwei Hashfunktionen auszuwerten und zwei Zugriffe in ein Feld durchzuführen.

In [CHM97] wird gezeigt, wie man diesen Ansatz auf r-Graphen, $r \geq 3$, ausdehnen kann. Weiter wird gezeigt, dass für $r = 3$ der Platzbedarf am

geringsten ist: Die Autoren geben als untere Platzschranke $1.222n$ an. Allerdings wird die Auswertung komplizierter: Man muss dann r Hashfunktionen auswerten und r Speicherzugriffe durchführen.

Pagh stellt ein Verfahren vor, das es erlaubt, eine MPHF unter Verwendung von $(2+\varepsilon)n$ zusätzlichen Wörtern zu konstruieren [Pag99]. Dietzfelbinger und Hagerup verbessern dieses Verfahren, so dass $(1+\varepsilon)n$ Wörter ausreichen [DH01]. Zur Auswertung der MPHF müssen jeweils zwei einfache Hashfunktionen ausgewertet werden und ein Speicherzugriff durchgeführt werden. Im Gegensatz zu [CHM92, CHM97] wird bei [Pag99] und [DH01] auf die Annahme, dass die verwendeten Hashfunktionen volle Unabhängigkeit gewährleisten, verzichtet.[1]

Unser Verfahren arbeitet ähnlich wie das in [CHM92, CHM97] vorgestellte. Unter der Annahme der vollen Unabhängigkeit der gewählten Hashfunktionen benötigt es $(1.16+\varepsilon)n$ viele zusätzliche Wörter aus $[n]$. Durch Einführung des Undo-One-Schrittes erreichen wir, dass etwa $(0.93+\varepsilon)n$ Wörter ausreichen. Der Aufwand für die Auswertung ist der gleiche wie bei [CHM92, CHM97].

Zu bemerken ist an dieser Stelle, dass MPHF bekannt sind, die bei konstanter Auswertezeit weniger Platz benötigen. Man bedenke, dass die hier vorgestellten Verfahren alle $O(n \log n)$ Bits Platz benötigen. Schmidt und Siegel [SS90] zeigen, dass $O(n + \log_2 \log_2 |U|)$ Bits reichen und Hagerup und Tholey [HT01] zeigen, wie man mit $n \cdot \log_2 e + \log_2 \log_2 |U| + O(\log_2 n)$ Bits Platz auskommt. Allerdings ist die Auswertung der MPHF in beiden Fällen viel komplizierter und aufwändiger als bei den anderen Verfahren.

2.1 Der Algorithmus

Sei eine Schlüsselmenge S mit $|S| = n$ gegeben. Sei weiter $m = \frac{2}{d}n$ für eine Konstante d.[2] Wir stellen zunächst zwei Möglichkeiten vor, einen „Zufallsgraphen“ G aus S zu erzeugen, um danach die Knoten dieses Graphen geeignet zu beschriften.

[1] Wenn man beim Verfahren aus [Pag99] die volle Unabhängigkeit der Hashwerte annimmt, kann man einen Platzbedarf von $\approx 1.44n$ zeigen [Die].

[2] Für die in der Einleitung zu diesem Kapitel besprochene Konstante c gilt dann: $c = \frac{2}{d}$. Die Größe d bezeichnet den durchschnittlichen Knotengrad des später aus S erzeugten Graphen.

2.1.1 Erzeugung des Zufallsgraphen

In der ersten Phase unseres Verfahrens wollen wir einen Graphen $G = ([m], E)$ aufbauen, dessen Kantenmenge E die Schlüsselmenge S repräsentiert.

Der naive Ansatz

Ein erster Ansatz wäre, wie in [CHM92] so lange zufällig zwei Hashfunktionen h_0 und h_1 aus einer geeigneten Klasse $\mathcal{H}$ von Hashfunktionen, die sich vollkommen zufällig verhalten, zu wählen, bis in der Menge

$$E = \{\{h_0(x), h_1(x)\} \mid x \in S\} \tag{2.1}$$

keine Kanten doppelt vorkommen, das heißt für alle $x, y \in S$ mit $x \neq y$ gilt $\{h_0(x), h_1(x)\} \neq \{h_0(y), h_1(y)\}$, und außerdem keine Schleifen auftreten, das heißt für alle $x \in S$ gilt: $h_0(x) \neq h_1(x)$ (siehe ERZEUGE-GRAPH in Algorithmus 2.1). Wenn E diese beiden Eigenschaften erfüllt, wollen wir h_0 und h_1 als *gutes Paar* bezeichnen.

```
ERZEUGE-GRAPH (S)
1  repeat
2     m := ⌈(2/d) n⌉
3     Wähle h_0, h_1 : U → [m] zufällig aus H
4     E := {{h_0(x), h_1(x)} | x ∈ S}
5     G := ([m], E)
6  until G hat keine Schleife und keine Doppelkante
7  return G
```

Algorithmus 2.1: Erzeugen eines Graphen aus einer Schlüsselmenge S

Allerdings stellen wir fest, dass die Wahrscheinlichkeit dafür, dass h_0 und h_1 ein gutes Paar bilden, für $d > 1$ klein ist:

Lemma 1. *Seien $h_0, h_1 : S \to [m]$ Zufallsfunktionen. Dann gilt:*

$$\lim_{n\to\infty} \textbf{Prob}(h_0 \textit{ und } h_1 \textit{ bilden ein gutes Paar}) \leq e^{-\frac{d}{2}\left(1+\frac{d}{2}\right)}. \tag{2.2}$$

Beweis. Sei $G = ([m], E)$, $E = \{(h_0(x), h_1(x)) \mid x \in S\}$ ein Graph. Dann erhalten wir

$$\textbf{Prob}(G \text{ hat keine Schleifen}) = (1 - 1/m)^n \leq (1 - 1/m)^{\frac{d}{2}m} < e^{-\frac{d}{2}}. \tag{2.3}$$

Die Wahrscheinlichkeit dafür, dass keine Doppelkanten auftreten, können wir leicht wie folgt herleiten: Für m Knoten in einem Graphen gibt es $M =$

$\binom{m}{2}$ mögliche Kanten, die keine Schleifen sind. Des weiteren gibt es M^n Folgen $(\{h_1(x_i), h_2(x_i)\})_{1 \le i \le n}$, von denen $M(M-1)\dots(M-n+1)$ Folgen sämtlich paarweise verschiedene Elemente besitzen. Damit erhalten wir:

$$\begin{aligned}&\textbf{Prob}(G \text{ hat keine Doppelkante} \mid G \text{ hat keine Schleifen})\\ &= \frac{M(M-1)\dots(M-n+1)}{M^n} = \prod_{i<n}\left(1-\frac{i}{M}\right) \le e^{\sum_{i<n} i/m} = e^{\frac{n(n-1)}{m(m-1)}}\end{aligned} \tag{2.4}$$

und mit

$$\lim_{n\to\infty} e^{\frac{n(n-1)}{m(m-1)}} = e^{-(d/2)^2} \tag{2.5}$$

ergibt sich die Behauptung. □

Was dies bedeutet, wollen wir an einigen Zahlen verdeutlichen: Wenn $d = 2$ ist, hat der Graph genauso viele Knoten wie Schlüssel. Dann ist $\textbf{Prob}(h_0 \text{ und } h_1 \text{ bilden ein gutes Paar}) \le 0.135$. Möchten wir hingegen nur halb so viele Knoten wie Schlüssel haben, ist $d = 4$ und $\textbf{Prob}(h_0 \text{ und } h_1 \text{ bilden ein gutes Paar}) \le 0.00248$. Damit ist das Verfahren für große d nicht mehr praktikabel.

Wenn wenig Platz vorhanden ist

Für große d müssen wir uns etwas anderes einfallen lassen. Kanten der Form $\{y, y\}$ kann man leicht dadurch verhindern, indem man $h_0(x) = h'_0(x)$ und $h_1(x) = (h_0(x) + 1 + h'_1(x)) \bmod m$ für zwei zufällig gewählte $h'_0 \colon U \to [m]$ und $h'_1 \colon U \to [m-1]$ bildet. Allerdings hilft das nicht gegen Doppelkanten.

Deswegen verbessern wir die Erfolgswahrscheinlichkeit durch folgenden Trick: Wir wählen drei Funktionen $h, h_1, h_2 \colon U \to [m]$ zufällig und konstruieren eine Abbildung $b \colon [m] \to \{1,2\}$, so dass im Graphen $G = ([m], E)$ mit

$$E := \{\{h(x), h_{b(h(x))}\} \mid x \in S\} \tag{2.6}$$

keine Schleifen und Doppelkanten vorkommen.

Bemerkung 2. Der so erzeugt Graph ist ein Zufallsgraph. Denn durch die Konstruktion ist sicher gestellt, dass das entstehende Gebilde keine Schleifen oder Doppelkanten enthält: Damit sind die Kanten unter der Bedingung, dass keine Schleifen oder Doppelkanten auftreten, rein zufällig und voneinander unabhängig verteilt, sofern h, h_1 und h_2 Zufallsfunktionen sind (siehe Bemerkung 1).

Wenn man den Graphen auf diese Weise konstruiert, braucht man später bei der Auswertung der MPHF wie im naiven Ansatz auch lediglich zwei Hashfunktionen auswerten, muss aber pro Knoten ein zusätzliches Bit speichern. Außerdem sollte man bedenken, dass man beim naiven Ansatz die beiden Hashfunktionen parallel auswerten kann, während man hier die zweite Hashfunktion erst auswerten kann, wenn man $b(h(x))$ kennt.

Berechnung der Abbildung b Um diese Abbildung zu konstruieren, berechnen wir für jedes $i \in [m]$ zunächst seinen *Bucket*

$$B_i^h := \{x \in S \mid h(x) = i\}. \tag{2.7}$$

Um den gesuchten Graphen zu bekommen, beginnen wir mit einer leeren (Kanten-)Menge E. Für $i = 0, 1, \ldots, m-1$ testen wir der Reihe nach, ob für $r = 1, 2$ eine der beiden Mengen $\{\{i, h_r(x)\} \mid x \in B_i^h\}$ eine Schleife oder eine Doppelkante erzeugt. Eine Schleife entsteht, falls $i = h_r(x)$ für ein $x \in B_i^h$ gilt und eine Doppelkante entsteht, falls für ein $x \in B_i^h$ die Kante $\{i, h_r(x)\}$ bereits in E enthalten ist (TESTE-BUCKET in Algorithmus 2.2).

```
TESTE-BUCKET (i, r)
1   for x ∈ B_i^h do /* Doppelkanten mit anderen Buckets oder Schleifen? */
2     if i = h_r(x) or {i, h_r(x)} ∈ E then return false endif
3   endfor
4   for {x, y} ∈ B_i^h, x ≠ y do /* Doppelkanten im Bucket? */
5     if h_r(x) = h_r(y) then return false endif
6   endfor
7   return true
```

Algorithmus 2.2: Test, ob Bucket i bei $b(i) = r$ keine Schleife und keine Doppelkante erzeugt

Falls sowohl für $r = 1$ als auch für $r = 2$ eine Schleife oder eine Doppelkante erzeugt wird, beginnen wir erneut mit der Wahl von h, h_1 und h_2. Entstehen für kein r Schleifen oder Doppelkanten, wählen wir $b(i) \in \{1, 2\}$ zufällig, ansonsten wählen wir $b(i) := r$ für das r, das keine Schleifen oder Doppelkanten erzeugt. Schließlich werden alle Kanten $\{i, h_{b(i)}(x)\}$, $x \in B_i^h$, zu E hinzugefügt.

Wenn auf diese Weise jedes $b(i)$ ermittelt werden kann, haben wir den Graphen $G = ([m], E)$ gefunden (KONSTRUIERE-GRAPH-VERBESSERT in Algorithmus 2.3 auf der nächsten Seite).

```
KONSTRUIERE-GRAPH-VERBESSERT (S)
1  repeat
2     Wähle h, h_1, h_2 zufällig
3     E := ∅
4     error := false
5     for i := 0, ..., m − 1 do
6        B_i^h := {x ∈ S | h(x) = i}
7     endfor
8     for i = 0, ..., n − 1 do
9        Wähle r ∈ {1, 2} zufällig.
10       if TESTE-BUCKET(i, r) then
11          b(i) := r
12       else if TESTE-BUCKET(i, 3 − r) then
13          b(i) := 3 − r
14       else
15          error := true
16          Brich for-Schleife ab.
17       endif
18       E := E ∪ {{i, h_{b(i)}(x)} | x ∈ B_i^h}
19    endfor
20 until not error
21 return ([m], E)
```

Algorithmus 2.3: Aufbau eines Zufallsgraphen mit erhöhter Wahrscheinlichkeit aus einer Schlüsselmenge S

Analyse des Algorithmus KONSTRUIERE-GRAPH-VERBESSERT Wir wollen die erwartete Laufzeit von Algorithmus KONSTRUIERE-GRAPH-VERBESSERT abschätzten.

Lemma 2. *Mit Wahrscheinlichkeit* $1 - O(1/n)$ *wird die* ***repeat-until****-Schleife in Algorithmus* KONSTRUIERE-GRAPH-VERBESSERT *nur einmal durchlaufen. Die erwartete Laufzeit dieses Algorihtmus' ist* $O(n)$.

Beweis. Offensichtlich ist der Algorithmus genau dann erfolgreich, wenn für jedes $i \in [m]$ ein $b(i)$ bestimmt werden kann. Wir werden nun die Wahrscheinlichkeit dafür, dass ein $i \in [m]$ existiert, für das kein $b(i)$ bestimmt werden kann, abschätzen.

Sei $\mathcal{T} = (h, h_1, h_2)$ das Tripel der gewählten Funktionen. Damit der Algorithmus scheitert, muss eines der drei folgenden Ereignisse eintreten:

Bucket i erzeugt eine Schleife: Das ist genau dann der Fall, wenn es $x, y \in B_i^h$ gibt, so dass $h_1(x) = i$ und $h_2(y) = i$ gilt. Wenn wir dieses

Ereignis mit $X_i^{\mathcal{T}}$ bezeichnen, erhalten wir

$$X_i^{\mathcal{T}} \subseteq \bigcup_{\substack{x,y \in B_i^h \\ x \neq y}} \{h_1(x) = i \wedge h_2(y) = i\} \tag{2.8}$$

und

$$\mathbf{Prob}(X_i^{\mathcal{T}}) \leq |B_i^h|^2 \frac{1}{m^2}. \tag{2.9}$$

Bucket i erzeugt eine parallele Doppelkante: Dieser Fall kann nur eintreten, wenn es für beide $r \in \{1,2\}$ $x^{(r)}, y^{(r)} \in B_i^h$ mit $x^{(r)} \neq y^{(r)}$ gibt, so dass $h_r(x^{(r)}) = h_r(y^{(r)})$. Nennen wir dieses Ereignis $Y_i^{\mathcal{T}}$, können wir notieren:

$$Y_i^{\mathcal{T}} \subseteq \bigcup_{\substack{x,y \in B_i^h \\ x \neq y}} \bigcup_{\substack{x',y' \in B_i^h \\ x' \neq y'}} \{h_1(x) = h_1(y) \wedge h_2(x') = h_2(y')\} \tag{2.10}$$

und

$$\mathbf{Prob}(Y_i^{\mathcal{T}}) \leq |B_i^h|^4 \cdot \frac{1}{m^2}. \tag{2.11}$$

Bucket i erzeugt eine antiparallele Doppelkante: Dies kann nur passieren, wenn $l, l' < i$ mit $l \neq l'$ existieren, so dass Folgendes gilt (siehe Abbildung 2.1):

- Es gibt ein $x \in B_i^h$ und $y \in B_l^h$, so dass $h_1(x) = l$ und dass $h_{b_l}(y) = i$ gilt und
- es gibt ein $x' \in B_i^h$ und $y' \in B_{l'}^h$, so dass $h_2(x') = l'$ und dass $h_{b_{l'}}(y') = i$ gilt.

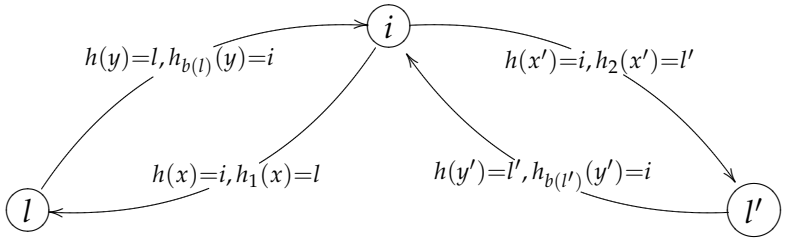


Abbildung 2.1: Illustration einer antiparallelen Kante

Wir wollen dieses Ereignis $Z_i^{\mathcal{T}}$ nennen. Damit erhalten wir

$$Z_i^{\mathcal{T}} \subseteq \bigcup_{l,l'<i} \bigcup_{x,x' \in B_i^h} \bigcup_{y \in B_l^h} \bigcup_{y' \in B_{l'}^h} \bigcup_{r,r' \in \{1,2\}} \{h_1(x) = l \wedge h_r(y) = i \wedge h_2(x') = l' \wedge h_{r'}(y') = i\} \quad (2.12)$$

und somit

$$Z_i^{\mathcal{T}} \subseteq \bigcup_{x,x' \in B_i^h} \bigcup_{y,y' \in S} \bigcup_{r,r' \in \{1,2\}} \{h_1(x) = h(y) \wedge h_r(y) = i \wedge h_2(x') = h(y') \wedge h_{r'}(y') = i\}, \quad (2.13)$$

und finden schließlich

$$\mathbf{Prob}(Z_i^{\mathcal{T}}) \leq |B_i^h|^2 \cdot n^2 \cdot \frac{4}{m^4} = d^2 |B_i^h|^2 \cdot \frac{1}{m^2}. \quad (2.14)$$

Um die erwartete Laufzeit von Algorithmus BUILD-GRAPH abzuschätzen, betrachten wir Stirlingzahlen der zweiten Art [GKP94, Abschnitt 6.1]. Eine Stirlingzahl der zweiten Art $\{{k \atop j}\}$ steht für die Anzahl der Möglichkeiten eine Menge mit k Elementen in j nichtleere Teilmengen aufzuteilen.

Lemma 3. *Für jedes feste k gilt:*

$$\mathbf{E}\left(\sum_{i<m} |B_i^h|^k\right) \leq n \sum_{j=0}^{k} \left\{ {k \atop j} \right\} \left(\frac{d}{2}\right)^{j-1} = O(n). \quad (2.15)$$

Beweis. Für ein festes k definieren wir

$$R^{(i)}_{h,x_1,\ldots,x_k} = \begin{cases} 1, & \text{falls } h(x_1) = \cdots = h(x_k) = i, \\ 0 & \text{sonst} \end{cases} \quad (2.16)$$

und

$$R_{h,x_1,\ldots,x_k} = \begin{cases} 1, & \text{falls } h(x_1) = \cdots = h(x_k), \\ 0 & \text{sonst.} \end{cases} \quad (2.17)$$

Es gilt [GKP94, Abschnitt 6.1, Gleichung (6.10)]:

$$|B_i^h|^k = \sum_{j=0}^{k} \left\{ {k \atop j} \right\} \binom{|B_i^h|}{j} j! \quad (2.18)$$

Bezeichnet $\langle x_1, \ldots, x_j \rangle \subseteq S$ eine j-elementige Teilmenge paarweise verschiedener Elemente von S, erhalten wir weiter

$$\sum_{i<m} |B_i^h|^k = \sum_{i<m} \sum_{j=0}^{k} \left\{ {k \atop j} \right\} \binom{|B_i^h|}{j} j! = \sum_{i<m} \sum_{j=0}^{k} \left\{ {k \atop j} \right\} j! \sum_{\langle x_1,\ldots,x_j \rangle \subseteq S} R^{(i)}_{h,x_1,\ldots,x_j}$$

$$= \sum_{j=0}^{k} \left\{ {k \atop j} \right\} j! \sum_{\langle x_1,\ldots,x_j \rangle \subseteq S} \sum_{i<m} R^{(i)}_{h,x_1,\ldots,x_j}$$

$$= \sum_{j=0}^{k} \left\{ {k \atop j} \right\} j! \sum_{\langle x_1,\ldots,x_j \rangle \subseteq S} R_{h,x_1,\ldots,x_j} \quad (2.19)$$

und

$$\mathbf{E}\Big(\sum_{i<m} |B_i^h|^k\Big) = \sum_{i<m} \sum_{h \in \mathcal{H}} \mathbf{Prob}(h \text{ ist gewählt}) \cdot |B_i^h|^k$$

$$= \sum_{h \in \mathcal{H}} \mathbf{Prob}(h \text{ ist gewählt}) \sum_{j=0}^{k} \left\{ {k \atop j} \right\} j! \sum_{\langle x_1,\ldots,x_j \rangle \subseteq S} R_{h,x_1,\ldots,x_j}$$

$$= \sum_{j=0}^{k} \left\{ {k \atop j} \right\} j! \sum_{\langle x_1,\ldots,x_j \rangle \subseteq S} \sum_{h \in \mathcal{H}} \mathbf{Prob}(h \text{ ist gewählt}) R_{h,x_1,\ldots,x_j}$$

$$= \sum_{j=0}^{k} \left\{ {k \atop j} \right\} j! \sum_{\langle x_1,\ldots,x_j \rangle \subseteq S} \mathbf{Prob}(R_{h,x_1,\ldots,x_j} = 1) = \sum_{j=0}^{k} \left\{ {k \atop j} \right\} j! \cdot \binom{n}{j} \left(\frac{1}{m}\right)^{j-1}$$

$$\leq \sum_{j=0}^{k} \left\{ {k \atop j} \right\} \cdot n^j \left(\frac{1}{m}\right)^{j-1} = \sum_{j=0}^{k} \left\{ {k \atop j} \right\} \left(\frac{d}{2}\right)^{j-1} n \quad (2.20)$$

□

Mit Lemma 3 erhalten wir als obere Schranke für die Wahrscheinlichkeit, dass eines der Ereignisse $X_i^{\mathcal{T}}, Y_i^{\mathcal{T}}, Z_i^{\mathcal{T}}$, $i = 0, \ldots, n-1$, für ein zufällig gewähltes h auftritt:

$$\mathbf{Prob}\left(\bigcup_{\mathcal{T}} \left\{ \mathcal{T} \text{ ist gewählt und } \bigcup_{i<m} (X_i^{\mathcal{T}} \cup Y_i^{\mathcal{T}} \cup Z_i^{\mathcal{T}}) \right\} \right)$$

$$\leq \sum_{\mathcal{T}} \left(\mathbf{Prob}(\mathcal{T} \text{ ist gewählt}) \cdot \frac{1}{m^2} \sum_{i<m} \Big((1+d^2)|B_i^h|^2 + |B_i^h|^4 \Big) \right)$$

$$= \frac{1}{m^2} \cdot O(n) = O(1/n). \quad (2.21)$$

Somit wird die **repeat**-Schleife von BUILD-GRAPH mit Wahrscheinlichkeit $1 - O(1/n)$ nur einmal durchlaufen. Wegen (2.20) ist die Laufzeit für eine solche Runde $O(n)$, wenn die Menge E mit Hilfe von n Listen implementiert wird. Damit ergibt sich als erwartete Laufzeit $O(n)$. □

2.1.2 Beschriftung der Knoten

Sei $G = (V, E)$, $V = [m]$, $|E| = n$, ein Graph mit $n = \frac{d}{2}m$ für eine Konstante d.[3] Wir wollen mit $\oplus$ und $\ominus$ die Addition und Subtraktion modulo n bezeichnen, das heißt

$$\begin{aligned} a \oplus b &= (a + b) \bmod n \\ a \ominus b &= (a - b) \bmod n \end{aligned} \tag{2.22}$$

Definition 2. Sei $G = (V, E)$ ein Graph und sei $g[u]$ die Beschriftung des Knotens $u \in V$. Für eine Kante $\{u, v\} \in E$ sei $\tilde{g}\colon E \to [m]$ eine Abbildung mit

$$\tilde{g}(\{u, v\}) := g[v] \oplus g[u] \tag{2.23}$$

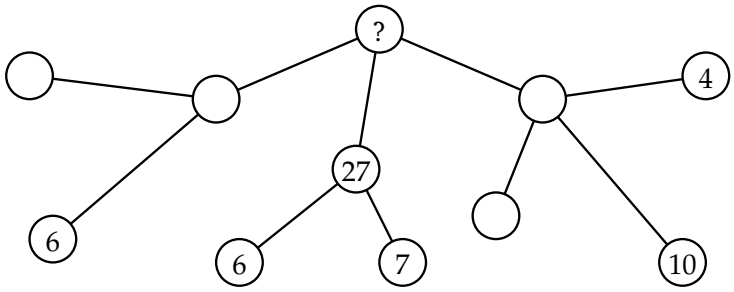
Diese Abbildung nennen wir *Kantenbeschriftung* von G.

Für jeden Knoten $u \in V$ möchten wir eine Beschriftung $g[u]$ derart finden, dass die Abbildung $\tilde{g}$ bijektiv ist. Um dies zu erreichen, wählen wir für jedes $u \in V$ die Beschriftung $g[u]$ zufällig aus $[n]$ so lange, bis entweder ein passendes $g[u]$ gefunden wird oder aber eine vorher festgelegte Anzahl von Versuchen überschritten wird.

Dabei merken wir uns nach jeder erfolgreichen Beschriftung eines Knotens, welche Werte im Bildbereich von $\tilde{g}$ durch die Beschriftung abgedeckt werden. Dadurch können wir leicht Kollisionen entdecken (siehe HAT-KOLLISION in Algorithmus 2.5 auf Seite 21). Als Datenstruktur benutzen wir dafür einfach den Bitvektor *used*.

Außerdem vermeiden wir stets, dass ein Knoten v die gleiche Beschriftung erhält wie einer der bereits beschrifteten Nachbarn eines Nachbars von v (siehe VERBOTEN in Algorithmus 2.4 auf Seite 21). Solche Werte nennen wir *verboten*. Wenn nämlich v mit einem verbotenen Wert beschriftet werden würde, kann die Beschriftung eines Nachbarn von v zu einem späteren Zeitpunkt nicht funktionieren, weil dann ein Knoten mit zwei gleich beschrifteten Nachbarn beschriftet werden müsste, was natürlich nicht funktioniert (siehe Abbildung 2.2 auf der nächsten Seite).

[3] Man beachte, dass die Buchstaben m und n ihre in der Graphentheorie üblichen Rollen getauscht haben: Der Graph hat m Knoten und n Kanten. Das ist der Tatsache geschuldet, dass jede Kante einem von n Schlüsseln entspricht.



Der Knoten ? darf nicht mit 4, 6 oder 10 beschriftet werden.

Abbildung 2.2: Beachten der Beschriftungen von Nachbarn unbeschrifteter Nachbarn

Definition 3. Einen Knoten, der zwei Nachbarn mit gleichen Beschriftungen besitzt, bezeichnen wir als *unmöglichen* Knoten.

Wir werden sehen, dass der Aufwand für das Betrachten der beschrifteten Nachbarn aller unbeschrifteten Nachbarn nicht allzu groß ist.

Um eine lineare Laufzeit zu erreichen, benutzen wir ein *Konto* (*account*), das mit $K \cdot n$ initialisiert wird. Von diesem Konto werden bei jedem Versuch, einen Knoten zu beschriften, die Kosten dieses Versuchs abgezogen. Mit Hilfe dieser Methode, den Aufwand in einem Konto zu verwalten, erreichen wir, dass Kosten für Knoten, die schwer zu beschriften sind, durch leicht zu beschriftende Knoten wett gemacht werden können.

Die Kosten eines Versuchs sind zum einen die Anzahl der zu berücksichtigenden Kanten des betrachteten Knotens. Weiterhin müssen wir für die Beschriftung von u den Aufwand für die Ermittlung verbotener Werte beachten. Dieser Aufwand beträgt höchstens

$$\sum_{\{u,v\}\in E} |\{\{v,v'\} \mid v' \in V\}| \tag{2.24}$$

Diese Vorgehensweise ist in BESCHRIFTE-KNOTEN (Algorithmus 2.6 auf der nächsten Seite) implementiert: Noch nicht beschriftete Knoten werden mit $\perp$ markiert. Bereits vergebene Werte aus dem Bildbereich von $\tilde{g}$ werden im Feld *used* markiert. Über die Größe von K werden wir uns später Gedanken machen.

Offensichtlich spielt die Reihenfolge, in der die Knoten zu beschriften sind, eine große Rolle: Je mehr Nachbarn eines Knotens v bereits beschriftet sind, um so schwerer ist eine Beschriftung von v zu finden, besonders dann, wenn v erst spät beschriftet wird. Hat v hingegen nur einen bereits

```
NACHBARN (u)
1   E_u := {w ∈ V | g[w] ≠ ⊥ ∧ {u,w} ∈ E}
2   return E_u

VERBOTEN (u)
1   M_u := {g[v'] | g[v'] ≠ ⊥ ∧ ∃v: g[v] = ⊥ ∧ {u,v} ∈ E ∧ {v,v'} ∈ E}
2   return M_u
```

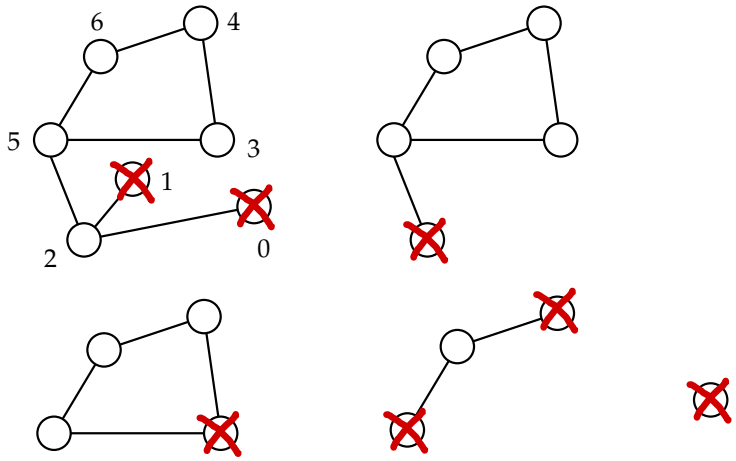
Algorithmus 2.4: Berechnung der verbotenen Werte für die Beschriftung des Knotens u

```
HAT-KOLLISION (t, E_u)
1   for v ∈ E_u do
2     account := account − 1
3     if used[t ⊕ g[v]] = true then return true endif
4   endfor
5   return false
```

Algorithmus 2.5: Test, ob die Beschriftung t des Knotens u eine Kollision erzeugt

```
BESCHRIFTE-KNOTEN (G)
1   account := Km
2   Fülle das Feld used[0,...,n−1] mit false.
3   Fülle das Feld g[0,...,m−1] mit ⊥.
4   for u ∈ V do
5     account := account − Σ_{{u,v}∈E} |{{v,v'} | v' ∈ V}|
6     while g[u] = ⊥ and account > 0 do
7       Wähle t ∈ [n] − VERBOTEN(u) zufällig.
8       if not HAT-KOLLISION(t, NACHBARN(u)) then
9         for v ∈ NACHBARN(u) do used[t ⊕ g[v]] := true endfor
10        g[u] := t
11      endif
12    endwhile
13  endfor
14  if ∃w ∈ [n]: g[w] = ⊥ then
15    error „G konnte nicht beschriftet werden!“
16  endif
```

Algorithmus 2.6: Beschriftungen g der Knoten des Graphen G finden



Es werden Schritt für Schritt Knoten kleinsten Grades aus dem Graphen entfernt. Anschließend werden die Knoten in umgekehrter Reihenfolge beschriftet; in diesem Beispiel wird also zuerst der Knoten 6 beschriftet und zuletzt der Knoten 0. Die Knoten mit den Nummern 3, 4, 5 und 6 bilden den 2-Kern des Graphen.

Abbildung 2.3: Schälen eines Graphen

beschrifteten Nachbarn, ist es leicht, für v selber eine Beschriftung zu finden.

Um eine gute Reihenfolge der Knoten zu finden, denken wir sukzessive jeweils einen Knoten mit kleinstem Grad von G abgeschnitten, so lange bis G keine Knoten mehr hat. Die Beschriftung soll dann in umgekehrter Reihenfolge erfolgen (Abbildung 2.3).

Durch dieses Abschälen des Graphen G erhalten wir auch dessen *k-Kerne*.

Definition 4. Der k-Kern eines Graphen G ist der größte Teilgraph G_k von G, dessen Knoten alle einen Grad von k oder mehr besitzen.

Ein $O(n+m)$-Zeit-Algorithmus zur Berechnung der k-Kerne ist leicht zu implementieren. Wir betrachten den Algorithmus KONSTRUIERE-KERNE (Algorithmus 2.7 auf der nächsten Seite), welcher die Listen L_k, $0 \leq k \leq \ell$, erzeugt, wobei L_k alle Knoten des k-Kerns ohne die des $k-1$-Kerns enthält. Darüber hinaus sind die Listen so beschaffen, dass die Verkettung von $L_0, \ldots, L_\ell$ die Reihenfolge angibt, in der die Knoten abgeschält wurden.

Das Unterprogramm KONSTRUIERE-k-KERN wird von KONSTRUIERE-KERNE aufgerufen, wenn nur noch Knoten vom Grad k oder größer im Graphen vorhanden sind, das heißt alle diese Knoten gehören zum k-Kern. Das Unterprogramm überprüft, ob es noch Knoten vom Grad k gibt; diese

Knoten gehören dann nicht zum $k+1$-Kern und gehören demzufolge zu L_k. Ein solcher Knoten wird aus dem Graphen gelöscht. Durch das Löschen kann es passieren, dass Knoten entstehen, deren Grad kleiner $l = k-1$ ist, diese gehören aber auch zu L_k. Durch das Löschen dieser Knoten können Knoten vom Grad $l = k-2$ entstehen, die wieder zu L_k gehören und so weiter. Dieses Problem wird durch rekursive Aufrufe des Unterprogramms gelöst.

Dass dieser Algorithmus wirklich eine Laufzeit von $O(n+m)$ hat, sieht man, indem man zählt, wie oft die Knoten in G angeschaut werden und wie oft das Unterprogramm KONSTRUIERE-k-KERN aufgerufen wird:

Jeder Knoten v wird wenigstens einmal betrachtet, aber höchstens $\deg(v)+1$ viele Mal. Alle nötigen Operationen (Kante löschen, Listenelemente löschen und einfügen) kosten $O(1)$ Schritte. Das Unterprogramm KONSTRUIERE-k-KERN wird nur dann aufgerufen, wenn ein Knoten gelöscht wird oder ein neuer k-Kern untersucht wird. Deswegen wird das Unterprogramm höchstens $2m$ mal aufgerufen, was die genannte Laufzeit liefert.

```
KONSTRUIERE-KERNE (G)
1   D_l := {v ∈ V | deg(v) = l}
2   k := 0
3   while E ≠ ∅ do
4      L_k sei eine leere Liste.
5      KONSTRUIERE-k-KERN(k, k)
6      k := k + 1
7   endwhile
8   return(L_0, ..., L_{k-1})
```

```
KONSTRUIERE-k-KERN (k, l)
1   while D_l ≠ ∅ do
2      Lösche einen Knoten u aus D_l
3      Hänge u an L_k an
4      for v ∈ {w ∈ V | {u, w} ∈ E} do
5         Hänge v hinten an L_k an.
6         grad := deg(v)
7         Verschiebe v aus D_grad in D_grad-1.
8         Lösche {u, v} aus E.
9      endfor
10     if l > 0 then
11        KONSTRUIERE-k-KERN(k, l − 1)
12     endif
13  endwhile
```

Algorithmus 2.7: Berechnung des k-Kerns eines Graphen G

Definition 5. Seien $L = (a_1, \ldots, a_r)$ und $L' = (b_1, \ldots, b_s)$ zwei Listen. Dann bezeichne $L \otimes L'$ die Verkettung von L und L':

$$L \otimes L' = (a_1, \ldots, a_r, b_1, \ldots, b_s). \tag{2.25}$$

Wir können nun den Algorithmus BESCHRIFTE-KNOTEN verbessern. Zunächst beschriften wir die Knoten der Liste $L := L_2 \otimes L_3 \otimes \cdots \otimes L_\ell$ in umgekehrter Reihenfolge. Nachdem das geschehen ist, haben wir es nur

noch mit Knoten zu tun, die jeweils höchstens einen beschrifteten Nachbarn haben. Um diese restlichen Knoten, die sich alle in L_1 befinden, zu beschriften, suchen wir im Feld *used* Stellen, die mit **false** belegt sind und beschriften die verbleibenden Knoten entsprechend. Und noch besser funktioniert dieser Schritt, wenn wir der Reihe nach alle Stellen im Feld *used*, die mit **false** belegt sind, abarbeiten und den nächsten noch nicht beschrifteten Knoten entsprechend beschriften. Dies haben wir im Algorithmus BESCHRIFTE-KNOTEN-VERBESSERT (Algorithmus 2.8 auf der nächsten Seite) implementiert.

2.1.3 Analyse des Algorithmus BESCHRIFTE-KNOTEN-VERBESSERT

Wir wollen annehmen, dass der Graph $G = ([m], E_G)$ mit $|E_G| = n = \frac{d}{2}m$ ein Zufallsgraph ist und die Kanten in G zufällig gewählt werden (siehe Bemerkung 1 auf Seite 6). Das heißt, G wird zufällig aus $\mathfrak{G}_{m,n}$ gewählt (siehe Abschnitt 1.5 auf Seite 8).

Die Kosten für einen Versuch, einen Knoten u zu beschriften, ergeben sich aus der Anzahl A der bereits beschrifteten Nachbarn von u sowie aus der Anzahl B der Nachbarn von noch nicht beschrifteten Nachbarn. Da jeder nicht beschriftete Nachbar v von u höchstens zwei beschriftete Nachbarn haben darf, hat v höchstens einen beschrifteten Nachbarn abgesehen von u. Das heißt, dass für u die Kosten für einen Beschriftungsversuch $A + B = \deg_G(u)$ betragen, wobei $\deg_G(u)$ der Grad des Knotens u im Graphen G ist.

Zunächst wollen wir die erwartete Laufzeit für die Graphen abschätzen, die die folgende Eigenschaft $g_p(G)$ besitzen: Jeder Knoten von G lässt sich mit einem Versuch mit einer Wahrscheinlichkeit $p > 0$ beschriften, unabhängig von den Beschriftungen der zuvor beschrifteten Knoten. Wir werden später angeben, wie man d wählen muss, damit $\mathbf{Prob}(g_p(G))$ positiv ist. Für einen Graphen $G \in \mathfrak{G}_{m,n}$, der die Eigenschaft $g_p(G)$ hat, seien T_G die Kosten, um G zu beschriften. Die Anzahl i der Versuche, bis der Knoten u beschriftet ist, ist eine geometrisch verteilte Zufallsgröße. Wir erhalten:

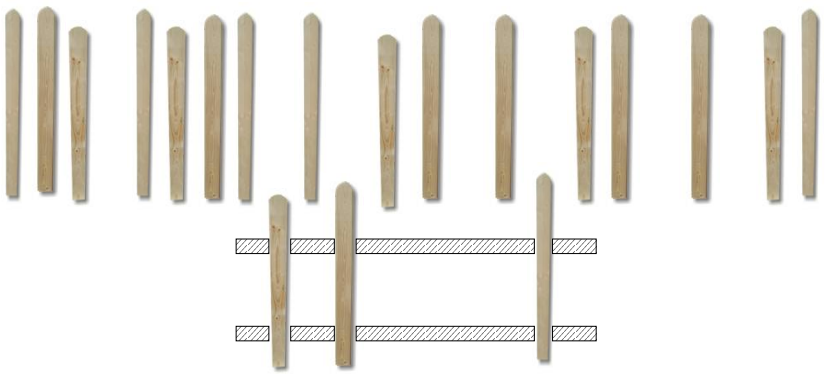
$$\mathbf{E}(T_G) \leq \sum_{u\in[m]} \left(\deg_G(u) \cdot \sum_{i\geq 0} p(i+1)(1-p)^i \right) = 2m \sum_{i\geq 0} p(i+1)(1-p)^i \tag{2.26}$$

Mit mit $\sum_{i\geq 0}(i+1)(1-p)^i = 1/p^2$ folgt:

$$\mathbf{E}(T_G) \leq \frac{2m}{p}. \tag{2.27}$$

```
BESCHRIFTE-KNOTEN-VERBESSERT (G)
1   (L_0, ..., L_ℓ) := KONSTRUIERE-KERNE(G)
2   (u_1, ..., u_k) := L_2 ⊗ L_3 ⊗ ··· ⊗ L_ℓ
3   account := Km
4   Fülle das Feld used[0, ..., n − 1] mit false.
5   Fülle das Feld g[0, ..., m − 1] mit ⊥.
6   for u = u_k, u_{k−1}, ..., u_1 do
7     account := account − Σ_{{u,v}∈E} |{{v, v'} | v' ∈ V}|
8     while g[u] = ⊥ and account > 0 do
9       Wähle t ∈ [m] − VERBOTEN(u) zufällig.
10      if not HAT-KOLLISION(t, NACHBARN(u)) then
11        for v ∈ NACHBARN(u) do used[t ⊕ g[v]] := true endfor
12        g[u] := t
13      endif
14    endwhile
15  endfor
16  if account > 0 then
17    for i := 0, ..., n − 1 do
18      while not used[i] do
19        u := Letztes Element von L_1
20        Lösche u von L_1.
21        if u hat einen beschrifteten Nachbarn v then
22          g[u] := i − g[v]
23          if g[u] < 0 then g[u] := g[u] + m endif
24          used[i] := true
25        else
26          Wähle g[u] ∈ [n] zufällig.
27        endif
28      endwhile
29    endfor
30    return g
31  else
32    error „G konnte nicht beschriftet werden!“
33  endif
```

Algorithmus 2.8: Verbesserte Variante der Knotenbeschriftung des Graphen G



Das Segment passt in den Gartenzaun mit Lücken. Das muss aber nicht immer so sein. Wie kann man sicherstellen, dass ein Segment mit 3 Latten garantiert passt?

Abbildung 2.4: Einfügen von Segmenten in einen Gartenzaun mit Lücken

Wenn man K genügend groß wählt, erhält man eine positive Wahrscheinlichkeit dafür, dass der Algorithmus erfolgreich ist, falls $g_p(G)$ wahr ist. Da die Zahl i aus (2.26) eine geometrisch verteilte Zufallsgröße ist, gilt $\mathbf{Var}(i) < p^{-2}$. Eine gute Wahl von *account* wäre somit

$$account = \frac{8m}{p} \geq \frac{2}{p}m + 2m \cdot 3\sqrt{\mathbf{Var}(i)} \tag{2.28}$$

das heißt, dass man zum Beispiel $K = 8/p$ wählen kann, um eine positive Wahrscheinlichkeit dafür zu erhalten, dass die Beschriftung des Graphen erfolgreich ist.

Wir müssen nun noch die beiden Wahrscheinlichkeiten p und $g_p(G)$ ermitteln. Dazu beobachten wir im Algorithmus BESCHRIFTE-KNOTEN-VERBESSERT die Anzahl r der bereits vergebenen Werte des Bildbereiches von $\tilde{g}$ zu dem Zeitpunkt, da der letzte Knoten aus L beschriftet wird.

Wir überlegen uns, wann die Wahrscheinlichkeit dafür, einen Knoten u mit einem Versuch zu beschriften, wenigstens p ist. Dazu betrachten wir folgendes Szenario: Wir stellen uns einen Gartenzaun vor, in dem ein paar Latten fehlen. Nun entschließen wir uns, den Zaun zu reparieren und gehen in den Baumarkt. Dort gibt es aber nicht genug einzelne Latten. Stattdessen gibt es Zaunsegmente, die wir in den Gartenzaun einfügen können (Abbildung 2.4). Jetzt stellt sich die Frage, wann ein Segment eingefügt werden kann und mit welcher Wahrscheinlichkeit es in eine zufällig gewählte Stelle passt.

Wir stellen uns einen Zaun mit n Plätzen für jeweils eine Latte vor. Diese Plätze sind im Kreis angeordnet. Außerdem sind bereits r Latten im Zaun.

Nun möchten wir ein Element mit k Latten einfügen. Die erste Latte des Segmentes könnte man in die $n-r$ Lücken platzieren, allerdings kann es dann sein, dass für die anderen $k-1$ Latten des Segmentes schon jeweils eine Latte im Zaun vorhanden ist. Eine Latte im Zaun versperrt aber höchstens $k-1$ freie Stellen im Zaun, in denen die erste Latte des Segmentes platziert werden kann. Das heißt, es werden von den $n-r$ freien Stellen des Zaunes höchstens $r(k-1)$ viele Stellen versperrt und es bleiben $n-r-r(k-1) = n-rk$ viele Stellen übrig, um das Segment zu platzieren. Eine solche Stelle findet man mit Wahrscheinlichkeit von mindestens

$$\frac{n-rk}{n} = 1 - \frac{rk}{n}. \tag{2.29}$$

Das heißt auch, dass man auf jeden Fall eine positive Wahrscheinlichkeit hat, eine freie Stelle zu finden, wenn $r < n/k$ gilt, also höchstens n/k Latten im Zaun sind beziehungsweise wenigstens $\frac{k-1}{k}n$ Lücken im Zaun sind. Diese Beobachtung wurde schon in [TY79] gemacht.

Die Beschriftung eines Knotens u im Graphen ist ein ebensolches Problem (siehe Abbildung 2.5 auf der nächsten Seite): Der Zaun ist der Bildbereich von $\tilde{g}$, also $[n]$, wobei bereits belegte Werte von $[n]$ (angekreuzte Zellen des Feldes in Abbildung 2.5 auf der nächsten Seite) die Zaunlatten darstellen. Das einzufügende Segment ergibt sich aus den bereits beschrifteten Nachbarn von u (mit einem Pfeil markiert). Die Beschriftung $g[u]$ bestimmt die Position des Segmentes im Zaun.

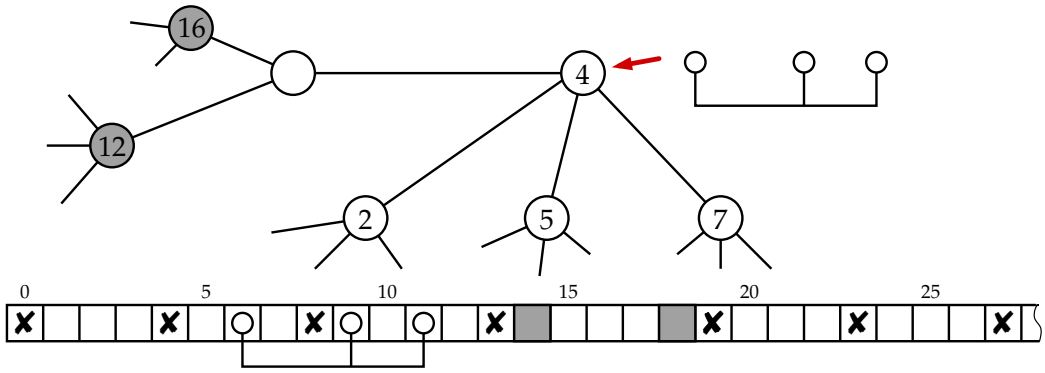
Weil wir außerdem verhindern wollen, dass auf Grund der Beschriftung von u ein unmöglicher Knoten entsteht, denken wir noch einige Latten in den Zaun eingefügt, und zwar an den Stellen, die verhindern, dass u eine verbotene Beschriftung annimmt (dunkel hinterlegte Zellen im Feld von Abbildung 2.5 auf der nächsten Seite). Das dies nicht allzu viele sind, wollen wir hier zeigen:

Lemma 4. *Die Wahrscheinlichkeit, dass ein Zufallsgraph G mit m Knoten und $n = \frac{d}{2}m$ Kanten einen Knoten mit Grad von mindestens $\ln m$ hat, ist höchstens $\frac{m^2}{e}\left(\frac{d}{\ln m}\right)^{\ln m}$. Insbesondere gilt für jedes $\ell > 0$:*

$$\textbf{Prob}(G \textit{ hat einen Knoten mit Grad} \geq \ln m) \in O(n^{-\ell}) \tag{2.30}$$

Beweis. Wir betrachten einen beliebigen Knoten u und berechnen mit Hilfe von (1.4)

$$\textbf{Prob}(u \text{ hat Grad von } \geq k) \leq \binom{n}{k}\left(\frac{2}{m}\right)^k \leq \frac{(en)^k}{ek^k}\frac{2^k}{m^k} = \frac{(de)^k}{ek^k} \tag{2.31}$$



Dieses Bild zeigt die Situation während der Beschriftung des mit dem Pfeil markierten Knotens. Angekreuzte Werte im Feld bedeuten bereits vergebene Werte des Bildbereiches von $\tilde{g}$. Die Beschriftungen 12 und 16 sind verboten. Damit werden im Bildbereich Werte (14 und 18) blockiert (grau hinterlegte Felder). Die bereits beschrifteten Nachbarn bilden ein Segment, dass in das Feld unten eingepasst werden muss. Die Beschriftung des markierten Knotens mit 4 ist zum Beispiel eine Beschriftung, durch die das Segment in das Feld eingepasst werden kann.

Abbildung 2.5: Beschriften der Knoten des Graphen G

Setzen wir $k = \ln m$, erhalten wir

$$\textbf{Prob}(G \text{ hat einen Knoten mit Grad} \geq \ln m) \leq \frac{m^2}{e}\left(\frac{d}{\ln m}\right)^{\ln m} \tag{2.32}$$

Für $\ln\ln m > 2 + \ln(d) + \ell$ gilt

$$m^2 \cdot m^{\ln(d)} \cdot \frac{1}{(\ln m)^{\ln m}} < m^{-\ell}, \tag{2.33}$$

woraus der zweite Teil der Behauptung folgt. □

Wir können also davon ausgehen, dass für alle Knoten u von G gilt: $\deg_G(u) \leq \ln m$. Weil jeder nicht beschriftete Knoten höchstens 2 beschriftete Nachbarn hat, gibt es für jeden Knoten weniger als $\ln m$ verbotene Beschriftungen und damit reicht es, sich höchstens $\ln m$ zusätzliche Zellen in *used* als verboten – und damit markiert – zu denken.

Seien zu dem Zeitpunkt, da der letzte Knoten der Liste L_k, $k \geq 2$ beschriftet wird, r_k Werte aus dem Bildbereich von $\tilde{g}$ bereits vergeben. Dann ist die Wahrscheinlichkeit für jeden Knoten u, mit einem Versuch beschriftet zu werden, wenigstens

$$\min_{k\geq 2}\left\{1 - \frac{k}{n}(r_k + \ln m)\right\}. \tag{2.34}$$

Und wenn

$$r_k \leq \frac{1-p}{k} n - \ln m, \tag{2.35}$$

gilt, haben wir für jeden Knoten eine Wahrscheinlichkeit von wenigstens

$$1 - \frac{(r_k + \ln m)k}{n} \geq 1 - \frac{\frac{1-p}{k} nk}{n} = p. \tag{2.36}$$

Analyse für $d < 1$

In [CHM92] wurde in der Analyse des Verfahrens benutzt, dass ein Zufallsgraph mit m Knoten und $n = \frac{d}{2} m$ Kanten mit konstanter Wahrscheinlichkeit keine Kreise enthält. Denn mit Graphen, die Kreise enthalten, konnte das dort beschriebene Verfahren nichts anfangen. Unser Verfahren jedoch kann auch Kreise beschriften.

Wir nehmen an, dass der Graph mit dem naiven Ansatz aus Abschnitt 2.1.1 auf Seite 12 entstanden ist. Wir zeigen, dass es für $d < 1$ nur wenige Schlüssel $x \in S$ gibt, deren Kanten auf einem Kreis liegen.[4] Sei c_ℓ die Anzahl von Kreisen der Größe $\ell \geq 3$ und k_ℓ die Anzahl von Kanten in Kreisen der Größe ℓ. Offensichtlich ist $\ell k_\ell < c_\ell$.

Sei $x_1, \ldots, x_\ell$ eine beliebige Folge von paarweise verschiedenen Schlüsseln. Diese Schlüssel bilden einen Kreis der Größe ℓ, wenn es $b_1, \ldots, b_\ell \in \{0,1\}$, gibt, so dass $h_{b_1}(x_1) = h_{1-b_2}(x_2)$ und $h_{b_2}(x_2) = h_{1-b_3}(x_3)$ und $\ldots$ und $h_{b_\ell}(x_\ell) = h_{1-b_1}(x_1)$ gilt.

Nun ist es egal, bei welchem Folgenglied ein solcher Kreis beginnt und in welche Richtung er läuft: Wenn die Folge $x_1, \ldots, x_\ell$ einen Kreis bildet für $2 \leq k \leq \ell$, dann tun dies auch die Folgen $x_k, x_{k+1}, \ldots, x_\ell, x_1, \ldots, x_{k-1}$ und $x_k, x_{k-1}, \ldots, x_1, x_\ell, \ldots, x_{k+1}$. Weil es weniger als m^ℓ Folgen mit ℓ verschiedenen Schlüsseln gibt, gilt:

$$\mathbf{E}(c_\ell) < n^\ell \cdot 2^\ell \cdot \frac{1}{m^\ell} \cdot \frac{1}{2\ell} = \frac{d^\ell}{2\ell} \tag{2.37}$$

Die erwartete Anzahl von Kreisen ist höchstens

$$\frac{1}{2} \sum_{l=3}^{\infty} \frac{d^\ell}{\ell} < -\frac{\ln(1-d)}{2} \tag{2.38}$$

und die erwartete Anzahl der Kanten in Kreisen ist höchstens

$$\frac{1}{2} \sum_{l=3}^{\infty} d^\ell < \frac{1}{2(1-d)}. \tag{2.39}$$

[4] Die Analyse der erwarteten Anzahl von Kanten in Kreisen in einem Zufallsgraphen folgt [Bol85, Abschnitt V.4]

Damit gilt

$$\textbf{Prob}(G \text{ hat wenigstens } \frac{2}{1-d} \text{ Kanten in Kreisen}) < \frac{1}{4} \tag{2.40}$$

Wenn G aber weniger als $D = \frac{2}{1-d}$ Kanten in Kreisen hat, ist $r_k = 0$ für $k > D$ und $r_k = O(1/n)$ für $2 \leq k \leq D$. Das heißt, für jedes feste $p < 1$ gilt, sofern n groß genug ist, die Bedingung (2.35); wir können uns das p frei wählen. Wir wählen $p = 0.99$. Wegen (2.40) ist dann $g_p(G) = 3/4$.

Analyse für $d \geq 1$

Für $d \geq 1$ lässt sich die Anzahl der Kanten, die auf Kreisen liegen, nicht mehr so leicht abschätzen. Auf jeden Fall steigt der Anteil dieser Kanten für wachsendes d.

Zunächst hilft uns ein wichtiges Resultat aus [PSW96]. In der genannten Arbeit wird gezeigt, dass mit hoher Wahrscheinlichkeit in einem Zufallsgraphen kein 3-Kern zu finden ist, sofern $d < 3.35$. Das heißt, dass mit hoher Wahrscheinlichkeit $r_k = 0$ gilt für $k \geq 3$. Wir müssen uns in diesem Fall lediglich um die Anzahl der Kanten kümmern, die vollständig im 2-Kern liegen. Das sind alle die Kanten, die zu keinem Knoten außerhalb des 2-Kerns inzident sind.

Sei $S_m(d)$, $d > 1$, die erwartete Anzahl von Kanten in einem Zufallsgraphen $G \in \mathfrak{G}_{m,q}$ für $q = d/m$, die wenigstens einen Knoten außerhalb des 2-Kerns eines Zufallsgraphen besitzen. Wir werden $S_m(d)$ im Abschnitt 2.2 bestimmen. Wir interessieren uns hier für die Funktion

$$s(d) := \frac{S_m(d)}{n} = \frac{2S_m(d)}{md}. \tag{2.41}$$

Genauer gesagt interessieren uns die Graphen G mit mindestens $(0.5 + \gamma)n$ Kanten außerhalb des 2-Kerns für irgendeine Konstante $\gamma > 0$. Denn dann ist $r_2 < (0.5 - \gamma)n$ und damit gilt nach (2.34) für die Wahrscheinlichkeit p für das Ereignis, für jeden Knoten u mit einem Versuch eine passende Beschriftung zu finden:

$$p > 1 - \frac{2r_2 + \ln(m)}{n} \geq 2\gamma - \frac{\ln(m)}{n} > \frac{3}{2}\gamma \tag{2.42}$$

für genügend große n.

Bleibt noch die Frage zu beantworten, wie groß die Wahrscheinlichkeit ist, einen Graphen auszuwählen, der mehr als $(0.5 + \gamma)n$ Kanten außerhalb des 2-Kerns besitzt.

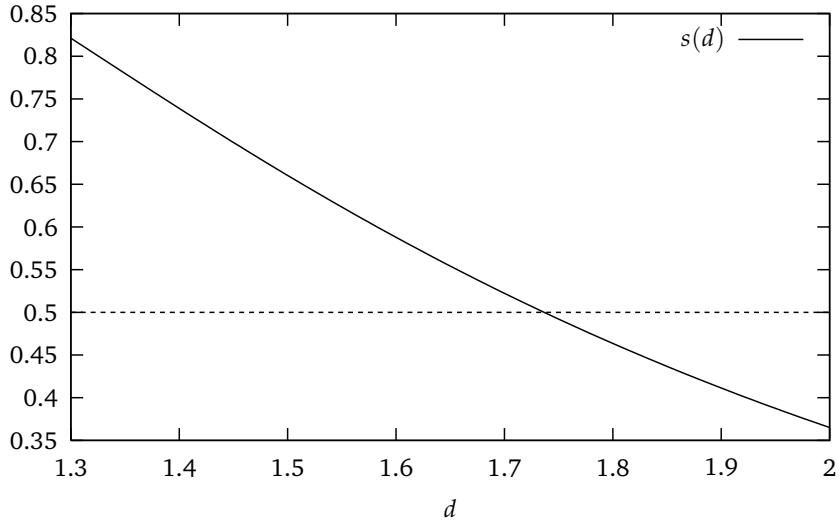


Abbildung 2.6: Verlauf von $s(d)$

Sei dazu d_0 so gewählt, dass $s(d_0) \geq 0.5 + \gamma$ gilt. Wir werden im nächsten Abschnitt sehen (Bemerkung 3 auf Seite 36), dass für $d = (1-\delta)d_0$ gilt:

$$\textbf{Prob}(G \text{ hat wenigstens } (0.5+\gamma)n \text{ Kanten außerhalb des 2-Kerns}) \geq \frac{3}{4},$$

wobei G zufällig unter den Graphen mit m Knoten und n Kanten, also im $\mathfrak{G}_{m,n}$-Modell, gewählt wird. Die Konstanten γ und δ können beliebig nahe an 0 sein. Damit ist auch $g_p(G) \geq 3/4$.

Der Verlauf der Funktion s ist in Abbildung 2.6 gezeigt. Zu guter Letzt ist die Frage zu klären, bei welchem $\tilde{d} = d$ diese Funktion den Wert 0.5 annimmt. Wir haben den Wert numerisch zu $\tilde{d} = 1.7366\ldots$ bestimmt. Damit kommt die so konstruierte MPHF mit $(c+\varepsilon)|S|$ Wörtern aus, wobei $c = 2/\tilde{d} = 1.152$. Damit können wir folgenden Satz formulieren:

Satz 1. *Sei $c \geq 1.152$. Der Algorithmus* BESCHRIFTE-KNOTEN-VERBESSERT *findet für eine geeignete Konstante $K = K(c)$ mit positiver Wahrscheinlichkeit eine Knotenbeschriftung $g\colon [m] \to [n]$ der $m = cn$ Knoten eines Zufallsgraphen $G = ([m], E)$ mit $|E| = n$, so dass die Abbildung $f\colon E \to [n]$ mit $f\colon \{u,v\} \mapsto (g(u) + g(v)) \bmod n$ bijektiv ist.*

2.2 Die erwartete Anzahl von Kanten außerhalb des 2-Kerns

Zunächst ermitteln wir die erwartete Anzahl der Kanten, die nicht im 2-Kern liegen, für das $\mathfrak{G}_{n,p}$-Modell und anschließend überlegen wir, weshalb wir mit diesem Erwartungswert auch im $\mathfrak{G}_{n,m}$-Modell rechnen können.

Lemma 5. *Sei $S_n(d)$, $d > 1$, die erwartete Anzahl von Kanten in einem Zufallsgraphen $G \in \mathfrak{G}_{n,p}$ für $p = d/n$, die wenigstens einen Knoten außerhalb des 2-Kerns von G besitzen. Dann gilt:*

$$S_n(d) = n \sum_{k=0}^{\infty} \frac{(k+1)^{k-1}}{k!} (de^{-d})^{k+1} - \frac{n}{d} \sum_{k=1}^{\infty} \frac{k^{k-2}(k-1)}{k!} (de^{-d})^k + O(1) \tag{2.43}$$

Beweis. In [Bol85] wird die erwartete Anzahl von Kanten, die in isolierten Bäumen liegen, hergeleitet. Wir wollen die gleiche Technik verwenden, um die erwartete Zahl von Kanten, die außerhalb des 2-Kerns liegen, herzuleiten.

Sei $G \in \mathfrak{G}_{n,p}$ ein Zufallsgraph mit $p = d/n$ für ein positives $d \neq 1$ und sei $q = 1 - p$. Sei weiterhin $S_n(d)$ die erwartete Anzahl von Kanten $\{u, v\}$, die außerhalb des 2-Kerns von G liegen, das heißt, bei denen einer der beiden Knoten u oder v außerhalb des 2-Kerns liegt.

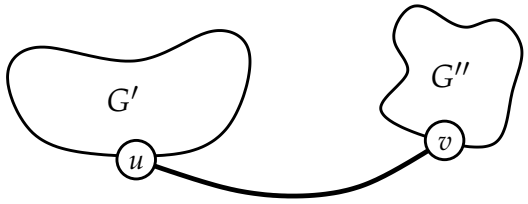


Abbildung 2.7: Illustration einer Kante

Sei $\{u, v\}$ eine Kante in G und seien G' und G'' die größten zusammenhängenden Teilgraphen, zu denen u beziehungsweise v gehören, wenn man die Kante $\{u, v\}$ aus G entfernt (siehe Abbildung 2.7).

Die Kante $\{u, v\}$ gehört genau dann nicht zum 2-Kern, wenn eines der folgenden Ereignisse eintritt:

(A_1) G' ist ist ein isolierter Baum, in dem v nicht enthalten ist und G'' ist kein isolierter Baum,

(A_2) G'' ist ein isolierter Baum, der u nicht enthält und G' ist kein isolierter Baum oder

(B) sowohl G' als auch G'' sind beide isolierte Bäume, die nicht v beziehungsweise nicht u enthalten.

Sei A' das Ereignis, dass G' ein isolierter Baum ist, der v nicht enthält. Offensichtlich ist dann $\mathbf{Prob}(A_1) = \mathbf{Prob}(A_2) = \mathbf{Prob}(A') - \mathbf{Prob}(B)$. Damit ist

$$S_n(d) = \binom{n}{2} \cdot p \cdot (2\,\mathbf{Prob}(A') - \mathbf{Prob}(B)) \\ = \underbrace{n(n-1) \cdot p \cdot \mathbf{Prob}(A')}_{=\alpha} - \beta, \quad (2.44)$$

wobei β die erwartete Anzahl von Kanten in isolierten Bäumen ist. Gemäß [Bol85, Abschnitt V.2] gilt

$$\beta = n \cdot \frac{1}{d} \sum_{k=1}^{\infty} \frac{k^{k-2}(k-1)}{k!} (de^{-d})^k + O(1). \quad (2.45)$$

Die Wahrscheinlichkeit dafür, dass G'' ein isolierter Baum mit $k+1$ Knoten (einschließlich v) ist, ist $\binom{n-2}{k}(k+1)^{k-1}p^k q^{\binom{k}{2}+(n-k-1)(k+1)-1}$, also gilt:

$$\alpha = n(n-1) \cdot p \cdot \sum_{k=0}^{n-2} \binom{n-2}{k} (k+1)^{k-1} p^k q^{\binom{k}{2}+(n-k-1)(k+1)-1} \\ = (n-1) \cdot \sum_{k=0}^{n-2} \underbrace{\binom{n-2}{k} (k+1)^{k-1} \cdot d \cdot \left(\frac{d}{n}\right)^k \left(1 - \frac{d}{n}\right)^{n(k+1)-k(k+5)/2-2}}_{=a_k}$$

Wir können eine obere Schranke für α_k angeben:

$$\alpha_k \leq \frac{(k+1)^{k-1}}{k!} d^{k+1} e^{-d(k+1)} e^{dk^2/(2n)+5dk/(2n)} \left(1 - \frac{d}{n}\right)^{-2}. \quad (2.46)$$

Wenn $k \leq \sqrt{n}$ ist, dann folgt

$$\alpha_k \leq \frac{(k+1)^{k-1}}{k!} (de^{-d})^{k+1} (1 + d'k^2/n) \left(1 - \frac{d}{n}\right)^{-2} \\ \leq \frac{(k+1)^{k-1}}{k!} (de^{-d})^{k+1} (1 + D'k^2/n) \quad (2.47)$$

für passende Konstanten d', D'. Ebenso können wir eine untere Schranke für α_k angeben:

$$\alpha_k \geq \frac{(k+1)^{k-1}}{k!}\left(1-\frac{k+2}{n}\right)^k d^{k+1}e^{-d(k+1)}e^{-dk^2/n}$$
$$\geq \frac{(k+1)^{k-1}}{k!}(de^{-d})^{k+1}(1-D''k^2/n) \quad (2.48)$$

für eine passende Konstante D''. Nach der Stirlingschen Formel (1.3) erhalten wir $k! \geq \sqrt{2\pi k}(k/e)^k$. Damit erhalten wir schließlich für $\ell = \lfloor \sqrt[3]{n} \rfloor$:

$$\left|\sum_{k=0}^{\ell}\alpha_k - \sum_{k=0}^{\ell}\frac{(k+1)^{k-1}}{k!}(de^{-d})^{k+1}\right|$$
$$\leq \max\{D', D''\}\sum_{k=0}^{\ell}\frac{k^2}{n}\frac{(k+1)^{k-1}}{k!}(de^{-d})^{k+1}$$
$$\leq \max\{D', D''\}\sum_{k=0}^{\ell}\frac{k(k+1)}{n}\frac{(k+1)^{k-1}}{k!}(de^{-d})^{k+1}$$
$$\leq \frac{\max\{D', D''\}}{n}\sum_{k=0}^{\ell}k\frac{(k+1)^k e^k}{k^k}\frac{1}{\sqrt{2\pi k}}(de^{-d})^{k+1}$$
$$\leq \frac{\max\{D', D''\}}{n}\sum_{k=0}^{\ell}k\frac{e\cdot e^k}{\sqrt{k}}(de^{-d})^{k+1}$$
$$\leq \frac{\max\{D', D''\}}{n}\sum_{k=0}^{\ell}\sqrt{k}(\underbrace{de^{1-d}}_{<1})^{k+1} = O(1/n) \quad (2.49)$$

Die Beziehung $de^{1-d} < 1$ folgt unmittelbar aus (1.1), wenn man dort $x := 1-d$ setzt.

Jetzt zeigen wir noch, dass große k sehr wenig zu α beitragen. Für $0 \leq k < n-2$ gilt:

$$\frac{\alpha_{k+1}}{\alpha_k} = \frac{\binom{n-2}{k+1}}{\binom{n-2}{k}}\cdot\frac{(k+2)^k}{(k+1)^{k-1}}\cdot\frac{d}{n}\cdot\left(1-\frac{d}{n}\right)^{n-\frac{(k+1)(k+6)-k(k+5)}{2}}$$
$$\leq \frac{n-2-k}{k+1}\cdot(k+1)\cdot\left(1+\frac{1}{k+1}\right)^k\cdot\frac{d}{n}\cdot\left(1-\frac{d}{n}\right)^{n-k-3}$$
$$\leq \left(1-\frac{k+2}{n}\right)\cdot e\cdot d\cdot e^{-d(1-(k+2)/n)}\left(1-\frac{d}{n}\right)^{-1}$$
$$= \underbrace{d(1-(k+2)/n)\cdot e^{1-d(1-(k+2)/n)}}_{<1}\left(1-\frac{d}{n}\right)^{-1} \leq \left(1-\frac{d}{n}\right)^{-1} \quad (2.50)$$

Die Beziehung $d(1-(k+2)/n)\cdot e^{1-d(1-(k+2)/n)} < 1$ folgt ebenfalls unmittelbar aus (1.1) mit $x := 1-d(1-(k+2)/n)$. Mit der Beziehung (2.46) und der Stirlingschen Formel (1.3) finden wir eine Schranke für α_ℓ:

$$\begin{aligned}\alpha_\ell &\le \frac{(\ell+1)^{\ell-1}}{\ell!}(de^{-d})^{\ell+1}\cdot O(1) \le \frac{(\ell+1)^{\ell-1}}{\sqrt{2\pi\ell}}\frac{e^\ell}{\ell^\ell}(de^{-d})^{\ell+1}\cdot O(1)\\ &\le \left(1+\frac{1}{\ell}\right)^{\ell-1}\frac{1}{\ell}(de^{1-d})^{\ell+1}\cdot O(1) = o(n^{-t}) \quad (2.51)\end{aligned}$$

für jedes $t>0$. Da $\left(1-\frac{d}{n}\right)^{-1} < n$ gilt, finden wir mit (2.50):

$$\sum_{k=\ell+1}^{n-2} a_k \le \alpha_\ell \sum_{i=0}^{n}\left(1-\frac{d}{n}\right)^{-1} \le a_k n^2 = o(n^{-t})n^2 = o(n^{-1}) \qquad (2.52)$$

Außerdem können wir eine obere Schranke für $\sum_{k=\ell+1}^{\infty}\frac{(k+1)^{k-1}}{k!}(de^{-d})^{k+1}$ angeben:

$$\begin{aligned}\sum_{k=\ell+1}^{\infty}\frac{(k+1)^{k-1}}{k!}(de^{-d})^{k+1} &\le \frac{1}{\sqrt{2\pi}}\sum_{k=\ell+1}^{\infty}\frac{(k+1)^{k-1}}{k^k\cdot\sqrt{k}}e^k(de^{-d})^{k+1}\\ &\le \frac{1}{\sqrt{2\pi}}\sum_{k=\ell+1}^{\infty}\frac{e\cdot e^k}{k^{3/2}}\cdot(de^{-d})^{k+1}\\ &\le \frac{1}{\sqrt{2\pi}}\sum_{k=\ell+1}^{\infty}\frac{1}{k^{3/2}}(de^{1-d})^{k+1}\\ \le \frac{1}{\sqrt{2\pi}}(de^{1-d})^{\ell+1}\underbrace{\sum_{k=0}^{\infty}\frac{1}{(k+\ell+1)^{3/2}}(de^{1-d})^{k+1}}_{=O(1)} &= o(n^{-1}), \quad (2.53)\end{aligned}$$

denn es gilt $(de^{1-d})^{\ell+1} = o(n^{-t})$ für jedes $t>0$.

Die letzten beiden Ungleichungen zusammen mit (2.49) ergeben schließlich die gewünschte Abschätzung

$$\begin{aligned}&\left|\sum_{k=0}^{n-2}\alpha_k - \sum_{k=0}^{\infty}\frac{(k+1)^{k-1}}{k!}(de^{-d})^{k+1}\right|\\ &\qquad\le \left|\sum_{k=0}^{\ell}\alpha_k - \sum_{k=0}^{\ell}\frac{(k+1)^{k-1}}{k!}(de^{-d})^{k+1}\right|\\ &\qquad + \sum_{k=\ell+1}^{n-2}a_k + \sum_{k=\ell+1}^{\infty}\frac{(k+1)^{k-1}}{k!}(de^{-d})^{k+1} = O(1/n), \quad (2.54)\end{aligned}$$

was uns schließlich zu (2.43) führt. □

Dieser Erwartungswert wurde für das $\mathfrak{G}_{n,p}$-Modell berechnet. Wir müssen uns noch überlegen, dass wir mit dieser Zahl auch im $\mathfrak{G}_{n,m}$-Modell rechnen dürfen.

Bemerkung 3. Für einen Graphen G sei $S(G)$ die Anzahl der Kanten, die wenigstens einen Knoten außerhalb des 2-Kerns von G besitzen. Für jede Konstante $\delta > 0$ gilt: Ist G ein Graph, der zufällig unter den Graphen mit n Knoten und $m = (1-\delta)dn/2$ Kanten gewählt wird, dann gilt für genügend große n:

$$\mathbf{Prob}(S(G) \geq S_n(d)) \geq \frac{3}{4}. \tag{2.55}$$

Wir wollen die Richtigkeit dieser Aussage begründen: Wir geben den Beweis nicht in allen Einzelheiten an, skizzieren aber die wichtigsten Schritte.

In [Bol85, Kapitel V, Satz 9] wird gezeigt, dass die Anzahl von Kanten in Bäumen in einem Graphen G, der aus $\mathfrak{G}_{n,p}$, $p = (1-\delta)d/n$, gewählt wird, um ihren Erwartungswert konzentriert ist. Sei nun Q die Eigenschaft eines Graphen G, dass $|S(G) - S_n((1-\delta)d)| < n^{2/3}$ gilt. Ganz analog zu dem genannten Beweis kann man zeigen, dass fast alle Graphen G aus $\mathfrak{G}_{n,p}$ die Eigenschaft Q besitzen, das heißt $\lim_{n\to\infty} \mathbf{Prob}(Q) = 1$.

Nun benutzen wir einen Satz aus der Theorie der Zufallsgraphen [Bol85, Kapitel II, Satz 2 (i)], dessen Aussage wir hier wiedergeben:

Satz 2 ([Bol85, Kapitel II, Satz 2 (i)])**.** *Sei Q eine Eigenschaft eines Graphen und gelte $pqN \to \infty$ mit $N = \binom{n}{2}$ und $q = 1-p$. Dann sind die folgenden Behauptungen äquivalent:*

(a) Fast jeder Graph in $\mathfrak{G}_{n,p}$ hat die Eigenschaft Q.

(b) Seien $x > 0$ und $\varepsilon > 0$ gegeben. Wenn n genügend groß ist, dann gibt es $l \geq (1-\varepsilon)2x(pqN)^{1/2}$ ganze Zahlen $M_1, \ldots, M_l$ mit

$$pN - x(pqN)^{1/2} < M_1 < \cdots < M_l < pN + x(pqN)^{1/2},$$

so dass $\mathbf{Prob}_{M_i}(Q) > 1-\varepsilon$ gilt für jedes i mit $1 \leq i \leq l$.

Dabei ist $\mathbf{Prob}_{M_i}(Q)$ die Wahrscheinlichkeit dafür, dass ein Graph G, der aus der Menge der Graphen mit n Knoten M_i Kanten gezogen wird, die Eigenschaft Q besitzt.

Für die oben genannte Eigenschaft Q gilt der Teil (a). Aus dem Teil (b) folgt dann, dass es für $x = 1$ und $\varepsilon = 1/4$ wegen $m = pN$ $l = \frac{3}{2}(mq)^{1/2}$ paarweise voneinander verschiedene Zahlen M_i, $1 \leq i \leq l$, mit

$m - (mq)^{1/2} < M_i < m + (mq)^{1/2}$ gibt, so dass $\mathbf{Prob}_{M_i}(Q) \geq 3/4$ gilt. Da $l = \frac{3}{2}(mq)^{1/2}$, muss es ein i geben, für das $\tilde{k} := M_i \geq m$ gilt (Schubfachprinzip).

Es gibt eine Konstante $\zeta > 0$, so dass für genügend große n gilt:

$$\frac{S_n((1-\delta)d)}{S_n((1-\delta/2)d)} \geq 1 + \zeta \tag{2.56}$$

Wenn also ein Graph G die Eigenschaft Q hat, gilt für genügend große n auch:

$$S(G) \geq S_n((1-\delta)d) - n^{2/3} > S_n((1-\delta/2)d). \tag{2.57}$$

Daraus folgt, dass $\mathbf{Prob}_{\tilde{k}}(S(G) > S_n((1-\delta/2)d)) \geq 3/4$.

Sei G_k die Menge der Graphen mit n Knoten und k Kanten. Sei $R(k,\ell)$ die Menge der Graphen $G \in G_k$, für die $S(G) \geq \ell$ gilt. Dann gilt:

$$\frac{|R(k+1,\ell+1)|}{|G_{k+1}|} \leq \frac{|R(k,\ell)|}{|G_k|}. \tag{2.58}$$

Denn aus jedem Graphen $G \in R(k+1,\ell+1)$ kann man $k+1$ mal eine Kante entfernen, wobei unter den verbleibenden k Kanten wenigstens ℓ außerhalb des 2-Kerns liegen. Somit kann man aus G $k+1$ Graphen $G' \in R(k,\ell)$ erzeugen. Jeder so erzeugte Graph wird höchstens $N-k$ mal gezählt, das heißt $|R(k,\ell)| \geq (k+1)|R(k+1,\ell+1)|/(N-k)$. Nun gilt auch $|G_{k+1}|/|G_k| = \binom{N}{k+1}/\binom{N}{k} = (N-k)/(k+1)$, woraus (2.58) folgt.

Aus (2.58) folgt, dass falls $\mathbf{Prob}_k(S(G) \geq \ell) > a$ gilt, auch $\mathbf{Prob}_{k-1}(S(G) \geq \ell - 1) > a$ wahr ist. Das heißt, dass aus $\mathbf{Prob}_{\tilde{k}}(S(G) > S_n((1-\delta/2)d)) \geq 3/4$ und aus $\tilde{k} - m \leq (mq)^{1/2}$ schließlich

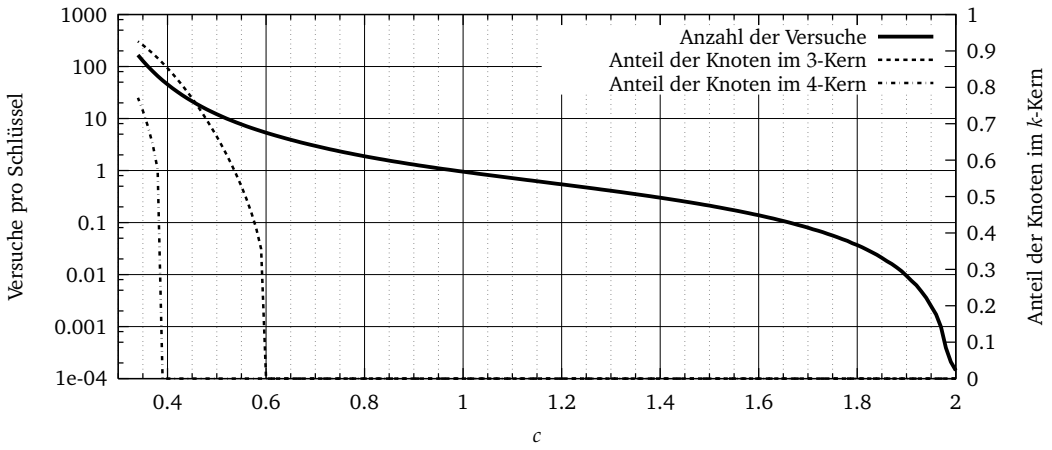
$$\mathbf{Prob}_m\left(S(G) > S_n((1-\delta/2)d) - (mq)^{1/2}\right) \geq \frac{3}{4} \tag{2.59}$$

folgt. Für genügend große n ist $S_n((1-\delta/2)d) - (mq)^{1/2} > S_n(d)$; damit folgt die Behauptung von Bemerkung 3.

2.2.1 Messungen

Das in diesem Abschnitt beschriebene Verfahren wurde in C++ implementiert. Es sollte untersucht werden, wie lange es in Abhängigkeit von c dauert, alle Knoten zu beschriften. Dazu wurden alle Versuche einen Knoten zu beschriften gezählt, ausgenommen der Versuche für den ersten Knoten einer jeden Zusammenhangskomponente. Denn diese Knoten haben zu dem

Zeitpunkt, da sie beschriftet werden, noch keine bereits beschrifteten Nachbarn. Für das Experiment wurden für jedes $c = 2.0, 1.99, \ldots, 0.32$ in jeweils fünfzig Runden 10^6 Schlüssel aus dem Bereich $[2^{28}]$ zufällig gewählt und für diese eine MPHF gesucht. Aus den in den jeweiligen Runden gezählten Versuchen ist für jedes c der Durchschnitt gebildet worden. Der Graph wurde mit KONSTRUIERE-GRAPH-VERBESSERT erzeugt. Die benutzen Hashfunktionen waren dabei Polynome dritten Grades.



Im Diagramm sind neben der gemessenen Anzahl von Versuchen (logarithmisch ausgerichtet an der linken Ordinate) auch der gemessene Anteil der Knoten im 3-Kern eingezeichnet (ausgerichtet an der rechten Ordinate).

Abbildung 2.8: Gemessene durchschnittliche Anzahl der benötigten Versuche, um die Knoten zu beschriften, in Abhängigkeit von c

Das Ergebnis des Experimentes ist in Abbildung 2.8 zu sehen: Allem Anschein nach ist es möglich, alle Knoten zu beschriften, wenn man $c = 0.33$ oder größer wählt, wobei der Rechenaufwand für $c \leq 0.4$ sehr groß (≈ 30 Versuche pro Schlüssel und mehr) ist. Bedenkt man, dass bei $c = 0.59$ und $c = 0.37$ ein 3- und ein 4-Kern auftreten [PSW96], verwundert es, dass die Kurve so glatt verläuft. Das Experiment legt nahe, dass das plötzliche Auftreten von Kernen nichts mit der Anzahl der benötigten Versuche zu tun hat.

2.3 Der Undo-One-Algorithmus zum Beschriften der Knoten

Wir werden den Algorithmus BESCHRIFTE-KNOTEN-VERBESSERT dahingehend weiter entwickeln, dass wir für dichtere Graphen als bisher zeigen können, dass sie sich beschriften lassen.

Allerdings hegen wir keine große Hoffnungen, dass dieser Algorithmus wirklich besser ist in dem Sinne, dass er schneller als der ursprüngliche wäre oder immer mit weniger Knoten auskäme. Der Grund für unseren Pessimismus sind Experimente [Hoc03]: Die analoge Vorgehensweise in [DH01] zur Verbesserung des Algorithmus aus [Pag99] brachte zwar eine theoretische Verbesserung des benötigten Platzes, in Experimenten jedoch zeigte sich keine Verbesserung. Der Grund ist, dass sich das Verfahren aus [Pag99] viel besser verhält, als es die Analyse vorhersagt.

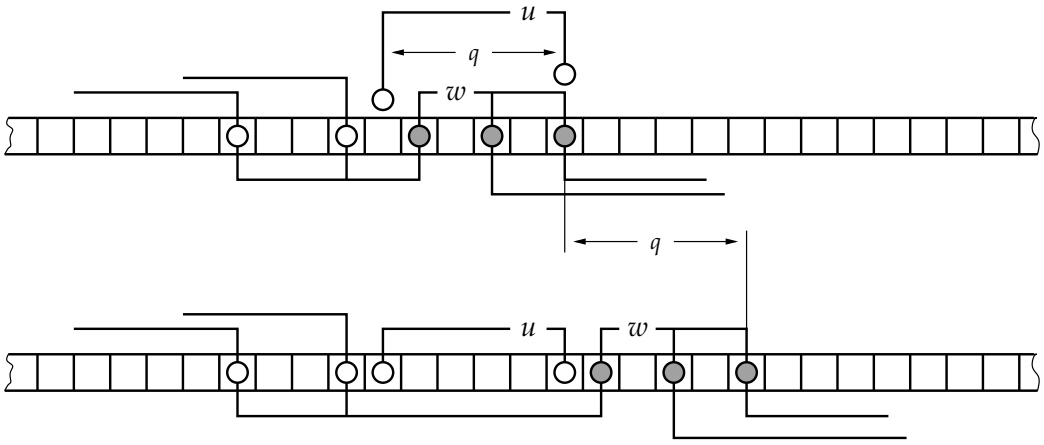
Wir wollen im Folgenden annehmen, dass n ungerade ist. Falls n gerade ist, entfernen wir eine beliebige Kante $\hat{e} \in E$ und erhalten so einen neuen Graphen G', den wir anstelle von G beschriften. Die Kantenbeschriftung, die wir für G' erhalten, sei $\tilde{g}'$. Wir wählen dann:

$$\tilde{g}(\{u,v\}) = \begin{cases} n-1, & \text{falls } \{u,v\} = \hat{e} \\ \tilde{g}'(\{u,v\}) & \text{sonst.} \end{cases} \tag{2.60}$$

2.3.1 Ein paar Vorüberlegungen

Wir verfolgen die in [DH01] vorgestellte Design- und Analysestrategie für das *Undo-one*-Prinzip: Wenn bei dem Versuch, einen Knoten mit genau zwei bereits beschrifteten Nachbarn zu beschriften, einer dieser Nachbarn eine Kollision verursacht, versuchen wir einen der beiden Knoten, die dafür verantwortlich sind, dass die Kollision auftritt, neu zu beschriften.

Betrachten wir dazu das Beispiel in Abbildung 2.9 und bedienen uns wie in Abschnitt 2.1.3 auf Seite 24 der bildhaften Vorstellung eines Gartenzauns: Der Knoten u soll beschriftet werden. Dessen zwei beschriftete Nachbarn sind bereits beschriftet, und zwar so, dass die Differenz der Beschriftungen q ist; dadurch ist ein Segment der Größe q beschrieben. Nun kann nach der zufällig gewählten Beschriftung $g[u]$ Folgendes geschehen: Eine Latte des Segmentes kann platziert werden, während die andere Latte mit dem bereits beschrifteten Knoten w kollidiert. In einer solchen Situation versuchen wir, den Knoten w mit $g[w] \oplus q$ zu beschriften: Wir verschieben das Segment w um q Positionen. Dabei bemerken wir, dass diejenigen Segmente, die zu beschrifteten Nachbarknoten von w gehören, beim Ver-



Der Knoten w erhält die neue Beschriftung $g[w] := g[w] \oplus q$.

Abbildung 2.9: Verschieben eines Knotens

schieben verändert werden. Mit ein bisschen Glück schaffen wir durch das Verschieben vom Segment w Platz für das Segment u.

Alternativ können wir versuchen, w mit $g[w] \ominus q$ zu beschriften oder zufällig eine vollkommen neue Beschriftung für w zu wählen.

Um zu klären, wann genau ein Segment w so verschoben werden kann, dass Platz für u entsteht, teilen wir den Bildbereich $[n]$ von $\tilde{g}$ in die zwei disjunkten Teilmengen $W = \{i \mid \text{Wert } i \text{ ist vergeben}\}$ und $B = [n] - W$ auf. Die Knoten in W wollen wir *weiß* färben und die Knoten in B *schwarz*. Dann betrachten wir den bipartiten Graphen $\mathcal{G} = (B, W, \mathcal{E})$ mit

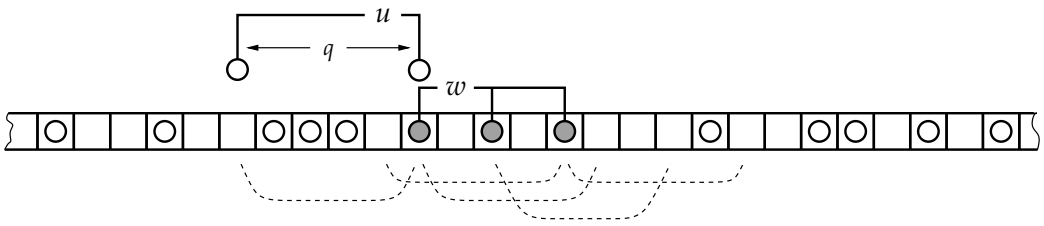
$$\mathcal{E} := \{\{i, j\} \mid i \in B, j \in W, i \oplus q = j\}. \tag{2.61}$$

Zunächst bemerken wir, dass $\mathcal{G}$ keine Kreise hat. Denn hätte $\mathcal{G}$ einen Kreis, dann hätte dieser eine gerade Länge, weil $\mathcal{G}$ bipartit ist. Dieser Kreis entspricht einer Untergruppe der additiven Gruppe $\mathbb{Z}_n$. Nach dem Satz von Lagrange ist die Größe dieser Untergruppe ein Teiler von n. Das kann aber hier nicht sein, da der Kreis eine gerade Anzahl von Knoten besitzt, n jedoch ungerade ist.

Für einen beschrifteten Knoten w sei $\mathcal{R}_w \subset [n]$ die Menge aller Werte des Bildbereiches von $\tilde{g}$, die durch w abgedeckt werden. $\mathcal{R}_w$ beschreibt also die Positionen im Zaun, die durch w ausgefüllt werden. Weiterhin sei

$$\mathcal{Z}_w := \{\{i, j\} \in \mathcal{E} \mid i \in \mathcal{R}_w, j \in W\} \tag{2.62}$$

die Menge der Kanten in $\mathcal{G}$, über die das Segment w verschoben werden kann. Wir wollen den Teilgraphen $\mathcal{G}_w$, der durch die Kantenmenge $\mathcal{Z}_w$ (und



Die grauen Knoten bilden die Menge $\mathcal{R}_w$ des Knotens w und die Linien unten sind die Menge $\mathcal{Z}_w$. Ändert man die Beschriftung von w in $g[w] \oplus q$, entsteht ein freier Platz, der es erlaubt u zu platzieren.

Abbildung 2.10: Illustration von $\mathcal{R}_w$ und $\mathcal{Z}_w$

allen zu $\mathcal{Z}_w$ inzidenten Knoten) bestimmt ist, als *Knotengraph* des Knotens w bezeichnen.

Definition 6. Ein Knotengraph $\mathcal{G}_w$ heißt *gut*, wenn $|\mathcal{Z}_w| \geq 2|\mathcal{R}_w| - 1$ gilt.

Da $\mathcal{G}$ kreisfrei ist, ist auch $\mathcal{G}_w$ stets kreisfrei, das heißt $\mathcal{G}_w$ besteht aus einer oder mehreren Zusammenhangskomponenten, die nichts anderes als einfache Pfade sind, in denen sich Knoten aus W und B abwechseln. Wir machen nun folgende Beobachtungen:

① Jeder Knoten in $\mathcal{G}_w$ hat einen Grad von höchstens 2. Damit ergibt sich, dass $|\mathcal{Z}_w| \leq 2|\mathcal{R}_w|$ gilt.

② Ist $\mathcal{G}_w$ ein guter Knotengraph (siehe Definition 6), gibt es keine Zusammenhangskomponente in $\mathcal{G}_w$, die mit einem schwarzen Knoten beginnt und endet. Ferner gibt es höchstens eine Zusammenhangskomponente, die mit einem schwarzen Knoten beginnt oder endet. Das heißt, wenn $\mathcal{G}_w$ ein guter Knotengraph ist, kann er in wenigstens eine Richtung um q Positionen verschoben werden.

③ Ist $\mathcal{G}_w$ ein guter Knotengraph, der aus wenigstens zwei disjunkten Pfaden besteht, beginnt und endet wenigstens einer dieser Pfade mit einem weißen Knoten. Deshalb entsteht beim Verschieben um q Positionen immer eine Stelle, um u zu platzieren (siehe Abbildung 2.11 auf der nächsten Seite).

④ Ist $\mathcal{G}_w$ ein guter Knotengraph, der nur eine Zusammenhangskomponente enthält, gibt es zwei Möglichkeiten: Entweder beginnt und endet dieser eine Pfad mit einem weißen Knoten, dann kann $\mathcal{G}_w$ wie in Beobachtung ③ um q in jede Richtung verschoben werden. Beginnt oder endet der Pfad allerdings mit einem schwarzen Knoten,

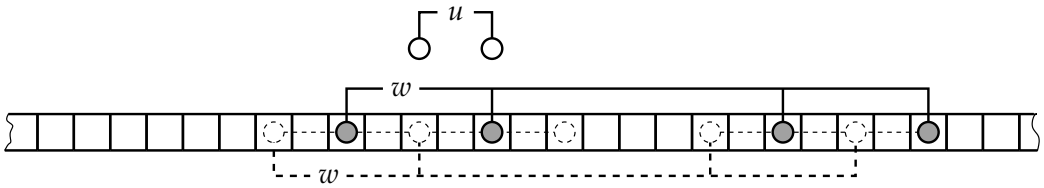


Abbildung 2.11: Illustration von Beobachtung ③

entsteht beim Verschieben kein Platz für u. Gibt es jedoch einen anderen guten Knotengraphen $\mathcal{G}_{w'}$, der eine Zusammenhangskomponente mit mindestens $|\mathcal{Z}_w|$ Kanten enthält, kann $\mathcal{G}_{w'}$ mit $\mathcal{G}_w$ vermischt werden (siehe Abbildung 2.12). Auf diese Art können immer zwei gute Knotengraphen vermischt werden, falls beide aus genau einer Zusammenhangskomponente bestehen.

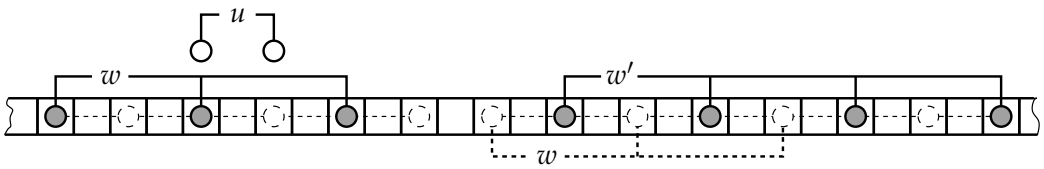


Abbildung 2.12: Illustration von Beobachtung ④

⑤ Durch das Verschieben eines Knotens w auf eine Position t kann es passieren, dass einer der unbeschrifteten Nachbarn von w ein unmöglicher Knoten (siehe Definition 3 auf Seite 20) wird. In diesem Fall darf w nicht mit t beschriftet werden.

2.3.2 Der Undo-One-Algorithmus

Der Algorithmus funktioniert ähnlich wie der Algorithmus BESCHRIFTE-KNOTEN-VERBESSERT (Algorithmus 2.8 auf Seite 25). Nach dem Berechnen der Listen $L_0, L_1, \ldots$ werden die Knoten aus $L_3, L_4, \ldots$ wie dort beschriftet. Die Knoten aus L_2 werden anders behandelt.

Sei u ein Knoten aus L_2. Zunächst raten wir eine Zahl t und versuchen u mit t zu beschriften. Wenn dieser Versuch erfolgreich ist, sind wir fertig. Ansonsten wählen wir unter den Knoten, mit denen u bei der Beschriftung t kollidiert, zufällig einen Knoten w aus, und führen eines der folgenden Zufallsexperimente durch, sofern w kein Nachbar von u ist:

Experiment 1: Wir versuchen, w mit $g[w] \oplus q$ und u mit t zu beschriften.

Experiment 2: Wir versuchen, w mit $g[w] \ominus q$ und u mit t zu beschriften.

Experiment 3: Wir versuchen, w mit einer beliebigen Zahl aus $[m]$ und u mit t zu beschriften.

Dabei ist q wieder der Abstand der zwei beschrifteten Nachbarn von u. Wenn das gewählte Experiment glückt, können wir u mit t beschriften und sind fertig. Ansonsten machen wir eventuelle Verschiebungen wieder rückgängig.[5]

Falls w ein Nachbar von u ist, führen wir kein solches Experiment durch, weil durch das Verschieben eines Nachbarn von u der Abstand q geändert wird.

Damit wir im Falle einer Kollision leicht ermitteln können, mit welchen Knoten u kollidiert, speichern wir im Feld *used* die Kante $\{s,t\}$, für welche $\tilde{g}(\{s,t\}) = i$ gilt, an der Position i ab. Ist i noch nicht abgedeckt, steht an der Stelle i im Feld *used* der Wert **false** (siehe Algorithmus FINDE-KOLLIDIERENDEN-KNOTEN in Algorithmus 2.9 auf der nächsten Seite).

Die Knoten außerhalb des 1-Kerns werden wie im Algorithmus BESCHRIFTE-KNOTEN-VERBESSERT beschriftet.

2.3.3 Analyse

Wir werden nun zeigen, dass die Wahrscheinlichkeit, einen Knoten mit einem Versuch zu beschriften, wenigstens p für eine Zahl $p > 0$ ist. Dazu wollen wir davon ausgehen, dass G keinen 3-Kern hat, was mit hoher Wahrscheinlichkeit der Fall ist. Da sich alle Knoten außerhalb des 1-Kerns leicht beschriften lassen, reicht es zu zeigen, dass für alle Knoten aus L_2 die Erfolgswahrscheinlichkeit wenigstens p ist.

Lemma 6. *Sei r die Zahl der Kanten außerhalb des 2-Kerns in G und μ eine Konstante, so dass*

$$2(1-\mu)r - 2(n-r) + m - |L_1| \geq \mu n \tag{2.63}$$

gilt und $r/n \geq \beta$ für eine Konstante β ist. Dann ist für jeden Knoten u aus L_2 die Erfolgswahrscheinlichkeit p, für u mit einem Versuch eine Beschriftung zu finden, positiv. Es gilt: $p \geq \min\{\mu\beta, \mu^2/48\}$.

[5] Man kann den Algorithmus auch variieren. Bevor man einen Knoten verschiebt, könnte man beispielsweise zunächst 10 mal eine Beschriftung t raten und erst dann eines der Zufallsexperimente durchführen. Aber das verbessert den Algorithmus nicht zwangsläufig dahingehend, dass die Erfolgswahrscheinlichkeit für die Beschriftung des gesamten Graphen verbessert wird.

```
BESCHRIFTE (u, t)
1   for v ∈ NACHBARN(u) do used[t ⊕ g[v]] := {u, v} endfor
2   g[u] := t

VERSCHIEBE (w, neu)
1   account := account − |NACHBARN(w)| − Σ_{{w,v}∈E} |{{v, v'} | v' ∈ V}|
2   alt := g[w]
3   for v ∈ NACHBARN(w) do used[alt ⊕ g[v]] := false endfor
4   g[w] := ⊥
5   if HAT-KOLLISION(neu, NACHBARN(w)) or neu ∈ VERBOTEN(w) then
6     BESCHRIFTE(w, alt)
7     return false
8   endif
9   BESCHRIFTE(w, neu)
10  return alt
```

Algorithmus 2.9: Verschieben und Beschriften eines Knotens

```
FINDE-KOLLIDIERENDEN-KNOTEN (u, t)
1   kollisionen := {used[t ⊕ g[v]] | v ∈ NACHBARN(u) ∧ used[t ⊕ g[v]] ≠ false}
2   mögliche_verschiebungen := ⋃ kollisionen
3   if mögliche_verschiebungen = ∅ then
4     return ⊥
5   else
6     return Ein zufällig gewähltes Element aus mögliche_verschiebungen
7   endif
```

Algorithmus 2.10: Kollidierende Knoten finden

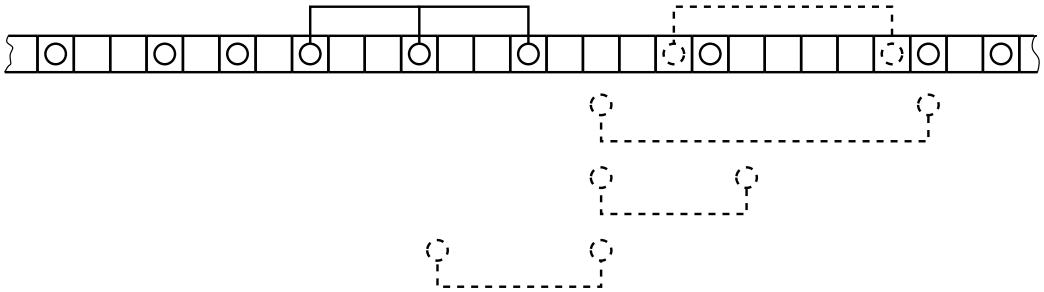
Beweis. Sei u ein beliebiger Knoten aus L_2. Wir betrachten für einen Moment die noch nicht beschrifteten Knoten aus $L_2 - \{u\} - \{v \mid \exists\{u, v\} \in E\}$. Diese können nach Beobachtung ⑤ die Ursache für die Entstehung unmöglicher Knoten sein, falls deren Nachbarn eine neue Beschriftung bekommen. Solche noch unbeschrifteten Knoten wollen wir als *blaue* Knoten bezeichnen. Durch irgendeine Beschriftung eines blauen Knotens v stellen wir jedoch sicher, dass v nicht mehr für die Entstehung eines unmöglichen Knotens verantwortlich ist.

Wir wählen die Beschriftung der blauen Knoten so, dass die entsprechenden Knotengraphen alle bis auf einen nicht gut sind (siehe Abbildung 2.13 auf Seite 46).

Dass durch die Beschriftungen der blauen Knoten Kollisionen entstehen, soll uns nicht weiter stören, wir interessieren uns vielmehr für den dadurch analog zu $\mathcal{G}$ entstehenden bipartiten Graphen $\mathcal{G}' = (B', W', \mathcal{E}')$.

```
BESCHRIFTE-KNOTEN-UNDOONE (G)
1   (L_0, ..., L_ℓ) := KONSTRUIERE-KERNE(G)
2   (u_1, ..., u_k) := L_3 ⊗ L_4 ⊗ ··· ⊗ L_ℓ
3   account := Km
4   Fülle das Feld used[0, ..., n − 1] mit false.
5   Fülle das Feld g[0, ..., m − 1] mit ⊥.
6   for u = u_k, u_{k−1}, ..., u_1 do
7     while g[u] = ⊥ and account > 0 do
8       account := account − Σ_{{u,v}∈E} |{{v, v'} | v' ∈ V}|
9       Wähle t ∈ [n] − VERBOTEN(u) zufällig.
10      if not HAT-KOLLISION(t, NACHBARN(u)) then BESCHRIFTE(u, t) endif
11    endwhile
12  endfor
13  (u'_1, ..., u'_{k'}) := L_2
14  for u = u'_{k'}, u'_{k'−1}, ..., u'_1 do
15    while g[u] = ⊥ and account > 0 do
16      Sei q der Abstand der Nachbarn von u.
17      account := account − Σ_{{u,v}∈E} |{{v, v'} | v' ∈ V}|
18      Wähle t ∈ [n] − VERBOTEN(u) zufällig.
19      w := FINDE-KOLLIDIERENDEN-KNOTEN(u, t)
20      if w = ⊥ then
21        BESCHRIFTE(u, t)
22      else if {u, w} ∉ E then
23        Wähle z ∈ {1, 2, 3} zufällig.
24        switch z
25          case 1 : neue_pos := g[w] ⊕ q
26          case 2 : neue_pos := g[w] ⊖ q
27          case 3 : neue_pos := Eine zufällige Zahl aus [n]
28        endswitch
29        alte_pos := VERSCHIEBE(w, neue_pos)
30        if not HAT-KOLLISION(t, NACHBARN(u)) then
31          BESCHRIFTE(u, t)
32        else if alte_pos ≠ false then
33          VERSCHIEBE(w, alte_pos)
34        endif
35      endif
36    endwhile
37  endfor
38  Weiter wie ab Zeile 15 in BESCHRIFTE-KNOTEN-VERBESSERT auf Seite 25
```

Algorithmus 2.11: Undo-One-Algorithmus zur Knotenbeschriftung des Graphen G



Hier sind die Knotengraphen der blauen Knoten (gestrichelt) für $q = 2$ illustriert. Die Knotengraphen der blauen Knoten werden so platziert, dass sie – bis auf einen – nicht gut sind. Das wird dadurch erreicht, dass der Knotengraph des ersten blauen Knotens so platziert wird, dass durch ihn ein weißer Knoten s belegt wird, der eine Entfernung von genau $2q$ zu einem schwarzen Knoten s' hat. Die restlichen blauen Knoten werden so beschriftet, dass deren Knotengraphen jeweils den Knoten $(s + s')/2$ enthalten (Knotengraphen unter dem Feld).

Abbildung 2.13: Beschriften der blauen Knoten

Offensichtlich ist $B' \supseteq B$, $W' \subseteq W$ und $\mathcal{E}' \subseteq \mathcal{E}$. Das heißt, wenn man für einen Knoten w dessen Knotengraphen $\mathcal{G}'_w$ in $\mathcal{G}'$ verschieben und dadurch w die neue Beschriftung t geben kann, kann man $\mathcal{G}_w$ auch in $\mathcal{G}$ auf die Position t verschieben. Außerdem kann man in $\mathcal{G}'$ bis auf die Nachbarn der unbeschrifteten Nachbarn von u alle Knoten nur so verschieben, dass keine unmöglichen Knoten entstehen. Nach Lemma 4 auf Seite 27 gibt es weniger als $(\ln m)^2$ viele beschriftete Nachbarn unbeschrifteter Nachbarn. Schließlich beobachten wir, dass $|W'| \geq r$ gilt. Wir unterscheiden zwei Fälle:

Fall 1: $|\mathcal{E}'| \leq 2(1-\mu)|W'|$**.** Sei

$$\mathcal{E}'' := \{\{i, j\} \mid i, j \in W', i \oplus q = j\}. \tag{2.64}$$

Jedes Element aus $\mathcal{E}''$ entspricht einer möglichen Beschriftung von u, die keine Kollisionen erzeugt. Sei $\mathcal{G}'' = (B', W', \mathcal{E}' \cup \mathcal{E}'')$ ein bipartiter Graph. Dann hat jeder Knoten aus W' in $\mathcal{G}''$ einen Grad von genau 2. Weil $\mathcal{E}'$ und $\mathcal{E}''$ disjunkt sind, folgt $2|W'| = |\mathcal{E}'| + 2|\mathcal{E}''|$ und

$$|\mathcal{E}''| = |W'| - |\mathcal{E}'|/2 \geq |W'| - (1-\mu)|W'| = \mu|W'|. \tag{2.65}$$

Damit ist die Wahrscheinlichkeit, eine Beschriftung für u mit einem Versuch zu finden, wenigstens $\frac{\mu|W'|}{n} \geq \frac{\mu r}{n} \geq \mu\beta$.

Fall 2: $|\mathcal{E}'| > 2(1-\mu)|W'|$**.** Wir wollen die Anzahl der guten Knotengraphen in $\mathcal{G}'$ bestimmen, die so verschoben werden können, dass keine unmöglichen Knoten entstehen.

Dazu stellen wir uns die Menge $\mathcal{E}'$ verdoppelt vor, also eine Multimenge $\tilde{\mathcal{E}}$, in der jedes Element aus $\mathcal{E}'$ genau zweimal vorkommt. Aus dieser Menge entfernen wir für jeden beschrifteten Knoten w die Kantenmenge $\mathcal{Z}'_w$ seines Knotengraphen $\mathcal{G}'_w$, aber nicht mehr als $2|\mathcal{R}'_w| - 2$ Kanten, wobei $\mathcal{R}'_w$ wieder die Menge der schwarzen Knoten in $\mathcal{G}'_w$ ist. Für jeden guten Knotengraphen verbleiben somit höchstens 2 Kanten in $\tilde{\mathcal{E}}$, für die anderen Knotengraphen hingegen verbleiben keine Kanten. Ist L die Menge der beschrifteten Knoten (einschließlich der blauen Knoten), bleiben wenigstens

$$\begin{aligned} 2|\mathcal{E}'| - \sum_{w\in L}(2|\mathcal{R}'_w| - 2) &\geq 4(1-\mu)|W'| - 2\sum_{w\in L}|\mathcal{R}'_w| + \sum_{w\in L} 2 \\ &\geq 4(1-\mu)|W'| - 4(n-r) + 2(m - |L_1|) \\ &\geq 4(1-\mu)r - 4(n-r) + 2(m-|L_1|) \geq 2\mu n. \end{aligned} \quad (2.66)$$

Kanten in $\tilde{\mathcal{E}}$ übrig. Weil alle Knotengraphen blauer Knoten bis auf höchstens einen nicht gut sind, gibt es wenigstens $\mu n - (\ln m)^2$ gute Knotengraphen, die in $\mathcal{G}'$ und damit auch in $\mathcal{G}$ so verschoben werden können, dass kein unmöglicher Knoten entsteht.

Jeder gute Knotengraph mit wenigstens zwei Zusammenhangskomponenten kann so verschoben werden, dass Platz für u entsteht. Für jede Position für u, die auf diese Weise entsteht, gibt es höchstens zwei Knotengraphen, die Platz schaffen, weil jeder schwarze Knoten von $\mathcal{G}$ in genau 2 Knotengraphen enthalten ist.

Gibt es wenigstens $\frac{\mu}{4}n$ gute Knotengraphen mit wenigstens zwei Zusammenhangskomponenten, können wenigstens $\frac{\mu}{8}n$ verschiedene freie Positionen geschaffen werden. Die Wahrscheinlichkeit, dass eine solche Stelle mit einer zufällig gewählten Beschriftung von u getroffen wird, ist damit $\frac{\mu}{8}$. Die Wahrscheinlichkeit dafür, dass in Zeile 23 von BESCHRIFTE-KNOTEN-UNDOONE das richtige z gewählt wird, ist $1/3$ und damit ist die Wahrscheinlichkeit einer erfolgreichen Beschriftung von u wenigstens $\frac{\mu}{24}$.

Gibt es hingegen weniger als $\frac{\mu}{4}n$ gute Knotengraphen mit wenigstens zwei Zusammenhangskomponenten, bleiben wenigstens $\frac{3\mu}{4}n - (\ln m)^2$ gute Knotengraphen mit genau einer Zusammenhangskomponente. Jeder von diesen kann, wie in Beobachtung ④ erklärt, mit einem anderen guten Knotengraphen mit genau einer Zusammenhangskomponente vermischt werden. Damit entstehen wenigstens $\frac{3\mu}{8}n - \frac{(\ln m)^2}{2}$ mögliche Positionen zum Platzieren von u. Für große

n ist dieser Ausdruck größer als $\frac{\mu}{4}n$. Die Wahrscheinlichkeit, dass durch die zufällige Wahl einer neuen Beschriftung eines Knotengraphen w mit genau einer Zusammenhangskomponente eine Stelle getroffen wird, in die w passt, ist wenigstens $\frac{\mu}{4}$. Die Wahrscheinlichkeit, dass für u eine Stelle gewählt wird, die einen solchen Knoten trifft und dass dieser erfolgreich neu platziert wird, ist demnach wenigstens $\frac{\mu^2}{16}$. Die Wahrscheinlichkeit dafür, dass in BESCHRIFTE-KNOTEN-UNDOONE $z = 3$ gewählt wird, ist $1/3$ und damit ist die Wahrscheinlichkeit einer erfolgreichen Beschriftung von u wenigstens $\frac{\mu^2}{48}$.

Damit ist die Wahrscheinlichkeit, mit einem Versuch eine erfolgreiche Beschriftung für u zu finden, durch $\min\left\{\mu\beta, \frac{\mu^2}{48}\right\}$, also durch eine Konstante, nach unten beschränkt. □

Schließlich bleibt noch zu klären, wann die Ungleichung (2.63) erfüllt ist. Die Anzahl der Knoten in L_1 kann man durch folgende Überlegung ermitteln: In L_1 befinden sich alle Knoten, die in isolierten Bäumen sitzen und die Knoten, die von Komponenten, die keine Bäume sind, abgeschält werden. Beim Abschälen dieser Knoten von G werden für jeden Baum mit s Knoten $s-1$ Kanten und für jeden Knoten, der nicht in einem isolierten Baum liegt, genau eine Kante entfernt. Ist T die Anzahl der Bäume in G, erhalten wir $|L_1| = r + T$. Nach Umstellen der Ungleichung (2.63) erhalten wir

$$\mu \leq \frac{3r - 2n\left(1 - \frac{1}{d}\right) + T}{n + 2r}. \tag{2.67}$$

Für r/n setzen wir wieder $s(d)$ ein. Für T finden wir in [Bol85]:

$$T = \frac{m}{d}\sum_{k=1}^{\infty}\frac{k^{k-2}}{k!}(de^{-d})^k + O(1) = \frac{2n}{d^2}\sum_{k=1}^{\infty}\frac{k^{k-2}}{k!}(de^{-d})^k + O(1). \tag{2.68}$$

Setzen wir dies alles in (2.63) ein, sehen wir, dass die Voraussetzung für Lemma 6 mit positiver Wahrscheinlichkeit erfüllt ist, wenn:

$$\mu \leq \frac{3s(d) - 2\left(1 - \frac{1}{d}\right) + \frac{2}{d^2}\sum_{k=1}^{\infty}\frac{k^{k-2}}{k!}(de^{-d})^k}{1 + 2s(d)}. \tag{2.69}$$

Den Zähler dieses Bruches wollen wir $\tilde{\mu}(d)$ nennen. Offenbar kann μ genau dann positiv gewählt werden, wenn $\tilde{\mu}(d)$ positiv ist. Anstatt $\tilde{\mu}(d)$ genau zu analysieren, betrachten wir Abbildung 2.14 auf der nächsten Seite. Im Bereich $1.5 \leq d \leq 2.5$ ist diese Funktion monoton.

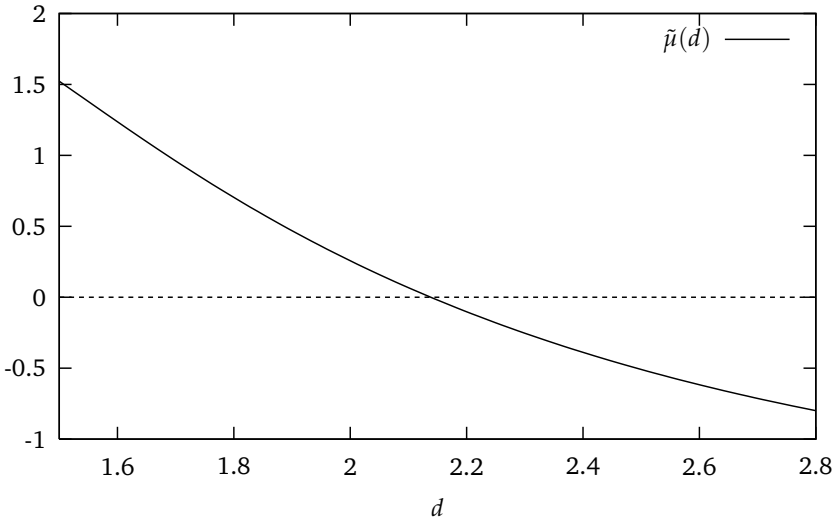


Abbildung 2.14: Verlauf von $\tilde{\mu}(d)$

Interessant ist nun die Nullstelle $\tilde{d}$ von $\tilde{\mu}(d)$. Numerisch kann man diese zu $\tilde{d} = 2.1383467\ldots$ bestimmen. Damit kommt die so konstruierte MPHF mit $(c+\varepsilon)|S|$ Wörtern aus, wobei $c = 2/\tilde{d} = 0.9353$.

Bemerkung 4. Die durch das Verschieben von Knoten zusätzlichen erwarteten Kosten betragen $O(R)$, wenn R die Anzahl von Versuchen ist, einen Knoten zu beschriften.

Dies sieht man analog zu Abschnitt 2.1.3 auf Seite 24: Bei jedem Versuch der Beschriftung des Knotens u müssen zusätzlich zu den in (2.27) ermittelten Kosten im Falle einer Verschiebung eines Knotens w die Kosten für die Inspektion der Nachbarn von w hinzugefügt werden. Für jeden unbeschrifteten Nachbarn v von w müssen außerdem dessen beschriftete Nachbarn betrachtet werden. Da v neben w nur einen weiteren bereits beschrifteten Nachbarn haben kann, erhält man $\deg_G(w)$ Kosten für den Versuch, w zu verschieben. Um die Verschiebung wieder rückgängig zu machen, hat man ebenfalls $\deg_G(w)$ Kosten. Weitere Kosten entstehen nicht.

Die Wahrscheinlichkeit, dass ein Knoten w zum Verschieben gewählt wird, ist höchstens $\deg_G(w)/n$. Wir bezeichnen die zusätzlichen Kosten für einen Verschiebeversuch im Graphen G mit T'_G und erhalten

$$\mathbf{E}(T'_G) = \sum_{w \in V} \mathbf{Prob}(w \text{ gewählt}) \cdot \deg_G(w) \leq \sum_{w \in [m]} \frac{\deg_G^2(w)}{n}. \qquad (2.70)$$

Wir interessieren uns nur für die Graphen $G \in \mathfrak{G}_{m,n}$, die die Voraussetzung

von Lemma 6 erfüllen. Diese Menge hängt von $\mu = \mu(d) = \mu(2/c)$ und von $\beta = \beta(d) = \beta(2/c)$ ab und sei mit $\mathfrak{G}_{c,n}$ bezeichnet. Für $c \geq 0.9353$ wissen wir, dass die Wahrscheinlichkeit dafür, dass ein zufällig gewählter Graph in $\mathfrak{G}_{c,n}$ liegt, positiv ist. Bezeichne nun T' die erwarteten zusätzlichen Kosten. Um $\mathbf{E}(T')$ zu ermitteln, betrachten wir das Wahrscheinlichkeitsexperiment, bei dem ein Graph G aus einer Menge zufällig gezogen wird[6] und die folgenden Ereignisse:

- Das Ereignis X_G ist, dass beim Ziehen aus der Menge $\mathfrak{G}_{m,n}$ der Graph G gewählt wurde.
- Das Ereignis Y_G ist, dass beim Ziehen aus der Menge $\mathfrak{G}_{c,n}$ der Graph G gewählt wurde.
- Das Ereignis Z ist, dass beim Ziehen aus der Menge $\mathfrak{G}_{m,n}$ ein Graph aus der Menge $\mathfrak{G}_{c,n}$ gewählt wurde. Wie bereits gesagt, ist $\mathbf{Prob}(Z) \geq \alpha$ für eine Konstante $\alpha > 0$. Weil sowohl die Anzahl von Kanten in Bäumen als auch die Anzahl der Kanten mit einem Knoten im 2-Kern stark um ihren Erwartungswert konzentriert sind (siehe Bemerkung 3 auf Seite 36), können wir annehmen, dass $\alpha \geq 3/4$.

Die Kosten für einen Graphen G sind $\mathbf{E}(T'_G)$. Nun gilt:

$$\begin{aligned}\mathbf{E}(T') &= \sum_{G\in\mathfrak{G}_{c,n}} \mathbf{Prob}(Y_G)\cdot\mathbf{E}(T'_G) = \sum_{G\in\mathfrak{G}_{c,n}} \mathbf{Prob}(X_G \mid Z)\cdot\mathbf{E}(T'_G)\\ &= \frac{1}{\mathbf{Prob}(Z)}\sum_{G\in\mathfrak{G}_{m,n}} \mathbf{Prob}(X_G)\cdot\mathbf{E}(T'_G) \leq \frac{1}{\alpha}\sum_{G\in\mathfrak{G}_{m,n}} \mathbf{Prob}(X_G)\cdot\mathbf{E}(T'_G)\\ &\leq \frac{1}{\alpha n}\sum_{G\in\mathfrak{G}_{m,n}} \mathbf{Prob}(X_G)\cdot\sum_{w\in[m]}\deg^2_G(w) \quad (2.71)\end{aligned}$$

Für einen Graphen $G = (V, E)$ und einen Knoten $w \in V$ sei

$$R^{(w)}_{G,e_1,e_2} = \begin{cases}1, & \text{falls sich } e_1, e_2 \in E \text{ in } w \text{ treffen,}\\ 0 & \text{sonst.}\end{cases}$$

Sei $\langle u, v\rangle \subseteq V = [m]$ eine Menge von zwei verschiedenen Knoten aus V

[6]Man beachte, dass wir es hier mit zwei Wahrscheinlichkeitsexperimenten zu tun haben: Das Ziehen eines Graphen und das anschließende Beschriften. Die Kosten für einen zufällig gezogenen Graphen G sind ein Erwartungswert, nämlich die erwarteten zusätzliche Kosten $E(T'_G)$.

und sei $V_w := V - \{w\}$. Dann gilt für einen festen Graphen G:

$$\sum_{w\in[m]} \deg_G^2(w) = \sum_{w\in[m]} \left(\deg_G(w) + 2\binom{\deg_G(w)}{2}\right)$$
$$= 2n + 2\sum_{w\in[m]} \sum_{\langle u,v\rangle\subseteq V_w} R^{(w)}_{G,\{u,w\},\{v,w\}}. \quad (2.72)$$

Setzen wir dies in die rechte Seite von (2.74) ein, erhalten wir

$$\alpha\,\mathbf{E}(T') < 2 + \frac{2}{n}\sum_{G\in\mathfrak{G}_{m,n}} \mathbf{Prob}(X_G)\cdot \sum_{w\in[m]}\sum_{\langle u,v\rangle\subseteq V_w} R^{(w)}_{G,\{u,w\},\{v,w\}}$$
$$= 2 + \frac{2}{n}\sum_{w\in[m]}\sum_{\langle u,v\rangle\subseteq V_w}\sum_{G\in\mathfrak{G}_{m,n}} \mathbf{Prob}(X_G)\cdot R^{(w)}_{G,\{u,w\},\{v,w\}}. \quad (2.73)$$

Es gibt $\binom{M}{n}$ Graphen mit n Kanten, $M = \binom{m}{2}$. Darunter sind für alle $\langle u,v,w\rangle \subseteq V$ genau $\binom{M-2}{n-2}$ Graphen, bei denen die beiden Kanten $\{u,w\}$ und $\{v,w\}$ vorhanden sind, denn die restlichen $n-2$ Kanten können sich beliebig auf die verbleibenden $M-2$ möglichen Plätze für Kanten aufteilen. Damit können wir weiter rechnen:

$$\alpha\,\mathbf{E}(T') < 2 + \frac{2}{n}\sum_{w\in[m]}\sum_{\langle u,v\rangle\subseteq V_w} \frac{\binom{M-2}{n-2}}{\binom{M}{n}} = 2 + \frac{2}{n}m\binom{m-1}{2}\frac{n(n-1)}{M(M-1)}$$
$$< 2 + 2m\binom{m}{2}\frac{n}{M^2} = 2 + 2m\frac{2n}{m(m-1)} = 2 + 2d + O\left(\frac{1}{m}\right). \quad (2.74)$$

Lässt man also nur $R = \varrho m$ Beschriftungsversuche zu, steigen die erwarteten Kosten um $2(1+d)\varrho m/\alpha + O(1)$.

Die erwartete Zahl von Beschriftungsversuchen für einen Knoten ist $1/p$ ($p = p(c)$ aus Lemma 6, siehe auch Herleitung von Ungleichung (2.28)), wobei die Varianz dieser Zufallsgröße $1/p^2$ ist. Damit ist $R = 3m/(\alpha p)$ eine gute Wahl. Zusammen mit dem Aufwand aus Ungleichung (2.28) erhalten wir als Wahl für *account*:

$$account \geq \frac{8m}{p} + 3\cdot\frac{2(1+d)m}{\alpha p} = \frac{8 + 6(1+d)/\alpha}{p}m. \quad (2.75)$$

Wegen $\alpha \geq 3/4$ erhalten wir mit $K = (16+8d)/p$ eine positive Wahrscheinlichkeit für den Erfolg der Beschriftung.

Wir wollen das Ergebnis unserer Untersuchen über den Algorithmus BESCHRIFTE-KNOTEN-UNDOONE in einem Satz formulieren:

Satz 3. *Sei $c \geq 0.9353$. Der Algorithmus* BESCHRIFTE-KNOTEN-UNDOONE *findet für eine geeignete Konstante $K = K(c)$ mit positiver Wahrscheinlichkeit eine Knotenbeschriftung $g\colon [m] \to [n]$ der $m = cn$ Knoten eines Zufallsgraphen $G = ([m], E)$ mit $|E| = n$, so dass die Abbildung $f\colon E \to [n]$ mit $f\colon \{u, v\} \mapsto (g(u) + g(v)) \bmod n$ bijektiv ist.*

2.4 Fazit und offene Fragen

Nachdem wir das Verfahren von [CHM92, CHM97] leicht modifiziert haben, war es uns möglich, eine MPHF mit Platz cn, $c \geq 1.16$ zu konstruieren, die sehr einfach auszuwerten ist. Dabei haben wir auch gezeigt, wie man aus einer gegebenen Schlüsselmenge S mit hoher Wahrscheinlichkeit einen Zufallsgraphen G mit cn Knoten und $|S|$ Kanten erzeugen kann, der S repräsentiert.

Für die Analyse haben wir die Zahl der Kanten, die nicht im 2-Kern liegen, asymptotisch genau bestimmt.

Messungen deuten sogar an, dass unser Verfahren mit Platz $0.4n$ noch sehr gut funktioniert, während das ursprüngliche Verfahren mit hoher Wahrscheinlichkeit mit Platz $(2 - \varepsilon)n$ nicht mehr funktioniert.

Durch die Einführung des Undo-One-Schrittes konnten wir die Platzschranke noch weiter verbessern: Wir konnten zeigen, dass das Undo-One-Verfahren weniger als n Platz benötigt, nämlich $(0.93 + \varepsilon)n$ Platz.

Ziel weiterer Forschungen muss es sein, die experimentell ermittelten Platzschranken auch beweisen zu können. Weiterhin bleibt zu klären, warum in den Messungen das plötzliche Auftauchen von k-Kernen sich nicht in der Kurve für die Zahl von Beschriftungsversuchen niederschlägt.

Schließlich kann man untersuchen, ob es lohnenswert ist, unsere Verfahren auf Hypergraphen zu verallgemeinern: Wie am Anfang dieses Kapitels erläutert, gelingt es in [CHM97] durch Benutzen von r-Graphen den Platzbedarf für $r = 3$ auf etwa $1.23n$ zu senken, wobei sich die Rechenzeit für eine Auswertung erhöht. Dabei wurden nur azyklische r-Graphen beschriftet und es wurde untersucht, wann r-Graphen kreisfrei sind. Wenn man zeigt, dass man analog zum Algorithmus KONSTRUIERE-GRAPH-VERBESSERT in Algorithmus 2.3 auf Seite 15 auch viele nicht kreisfreie r-Graphen beschriften kann, besteht die Hoffnung, die Platzschranke zu verbessern.

Kapitel 3

Dynamisches cachefreundliches Hashing

In diesem Kapitel wollen wir eine Datenstruktur für ein dynamisches Wörterbuch W vorstellen, das einen in W vorhandenen Schlüssel in einer vorab vereinbarten maximalen Zeit finden kann und sich zudem cachefreundlich verhält. *Cachefreundlich* soll heißen, dass die Operationen über dieser Datenstruktur gut für mehrschichtige Speicherarchitekturen geeignet sind.

Wir vereinbaren, dass in einem Feld $T[0,\ldots,m-1]$, der Größe m höchstens n Schlüssel aus einem Universum U gespeichert sind. Dabei ist $m \geq n$. Zellen, in denen kein Schlüssel gespeichert ist, haben den Eintrag $\perp$.

Dieses Kapitel ist der Hauptteil der vorliegenden Arbeit. Wir gehen wie folgt vor: Zuerst beschreiben wir bekannte Verfahren und zeigen deren Vor- und Nachteile auf. Anschließend beschreiben wir zwei neue cachefreundliche Verfahren, die sich allerdings sehr ähnlich sind. Eines davon analysieren wir hinsichtlich Platz- und Laufzeitbedarf. Nach der Analyse geben wir experimentelle Resultate an, und vergleichen die neuen Verfahren untereinander und mit den anderen Verfahren. Danach geben wir eine Möglichkeit an, wie man mit Hilfe von Fingerabdrücken unsere Verfahren laufzeiteffizient mit Zeichenketten benutzen kann. Im letzten Abschnitt fassen wir unsere Ergebnisse zusammen und geben einige offene Fragen an.

3.1 Bekannte Verfahren

Einen umfassenden Überblick über die wichtigsten Ansätze für dynamisches Hashing findet man in [PR04]. Die für unsere Arbeit interessanten wollen wir hier kurz vorstellen.

3.1.1 Lineares Sondieren

Das wohl simpelste Verfahren für ein dynamisches Wörterbuch ist *Lineares Sondieren* [Knu82, Abschnitt 6.4]: Um n Schlüssel zu speichern, benutzt man $m = (1+\varepsilon)n$ Zellen für ein $\varepsilon > 0$ sowie eine zufällig gewählte Hashfunktion $h\colon U \to [m]$. Soll ein Schlüssel x gespeichert werden, berechnet man $y = h(x)$ und sucht der Reihe nach unter den Zellen $T[y], T[y+1], \ldots, T[m-1], T[1], \ldots$ die erste freie und speichert x an dieser Position. Das Suchen eines Schlüssels x geht ähnlich: Man berechnet $y = h(x)$ und sucht der Reihe nach in den Zellen $T[y], T[y+1], \ldots, T[m-1], T[1], \ldots$ den Schlüssel x, bis man ihn entweder findet oder auf eine freie Zelle trifft.

Man sieht, dass man beim Suchen und Einfügen immer zusammenhängende Ketten von gefüllten Zellen betrachtet. Deswegen darf man beim Löschen eines Schlüssels x nicht einfach den Schlüssel entfernen, indem man die entsprechende Zelle mit $\perp$ markiert, denn das könnte eine Kette zerstören, wodurch man Schlüssel „verlieren“ könnte, sondern man muss gelöschte Felder gesondert markieren. Diese speziell markierten Felder muss man beim Einfügen und Suchen beachten. Das hat zur Folge, dass durch häufiges Löschen das Feld T mit als gelöscht markierten Feldern übersät wird. Als Alternative bietet [Knu82, Abschnitt 6.4] eine Prozedur für Löschen an, welche die gesonderte Markierung von gelöschten Zellen vermeidet, aber sehr aufwändig ist, weil dabei für die Schlüssel y, die durch das Löschen von x plötzlich nicht mehr in der Kette hängen, $h(y)$ bestimmt werden muss.

Eine wesentliche Schwäche des Verfahrens „Lineares Sondieren“ ist, dass das Einfügen oder Suchen eines Schlüssels sehr lange dauern kann, weil die Ketten sehr lang werden können: Für eine positive Suche ist die erwartete Anzahl besuchter Zellen $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$, und für die negative Suche und die Einfügeoperation ist die erwartete Anzahl besuchter Zellen $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ [Knu82, Abschnitt 6.4]. Dabei ist α der Auslastungsfaktor der Tabelle und steht mit ε in folgender Beziehung:

$$\alpha = \frac{1}{1+\varepsilon} \tag{3.1}$$

Für kleine ε macht man keinen großen Fehler, wenn man $1-\alpha \approx \varepsilon$ setzt. Dann erhält man für die erwartete Anzahl besuchter Zellen bei einer positiven Suche etwa $1/(2\varepsilon)$ und bei einer negativen Suche etwa $1/(2\varepsilon^2)$.

3.1.2 Cuckoo Hashing

Pagh und Rodler stellen 2001 *Cuckoo Hashing* vor, das unter Verwendung von $m = (2+\varepsilon)n$ Speicherzellen erlaubt, n Schlüssel zu speichern, wobei für eine Suche höchstens zwei Zellen gelesen werden müssen [PR01]. Außerdem hat Cuckoo Hashing – wie auch alle Verfahren, die auf diesem Prinzip beruhen – die Eigenschaft, dass die erwartete Einfügezeit konstant ist.

```
CUCKOO-INSERT (x)
1  for i ∈ {1,2} do if T_i[h_i(x)] = x then return endif endfor
2  i := 1
3  counter := ⌈3 ln m⌉
4  while x ≠ ⊥ and counter > 0 do
5     EXCHANGE(x, T_i[h_i(x)])
6     i := 3 − i
7     counter := counter − 1
8  endwhile
9  if counter = 0 then error „x konnte nicht eingefügt werden.“ endif
```

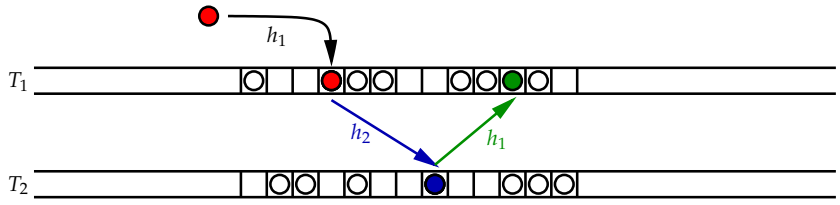
Algorithmus 3.1: Einfügen eines neuen Schlüssels beim Cuckoo Hashing

Zum Speichern der Schlüssel benutzt man zwei $r = m/2$ große Tabellen T_1 und T_2 und zwei zufällig gewählte Hashfunktionen $h_1, h_2 \colon U \to [r]$. Um einen Schlüssel x einzufügen, berechnet man $y_1 = h_1(x)$ und merkt sich $x_1 := T_1[y_1]$ und speichert x in $T_1[y_1]$. Ist $x_1 = \perp$, ist man fertig. Ansonsten berechnet man $y_2 = h_2(x_1)$ und merkt sich $x_2 := T_2[y_2]$ und speichert x_1 in $T_2[y_2]$. Ist $x_2 = \perp$, ist man fertig. Ansonsten fährt man fort und berechnet $y_3 = h_1(x_2)$ und so weiter. Analysiert man dieses Verfahren, stellt man fest, dass mit hoher Wahrscheinlichkeit $3 \ln m$ solcher Tauschoperationen ausreichen, bis eine freie Stelle gefunden ist (siehe CUCKOO-INSERT in Algorithmus 3.1).

3.1.3 d-äres Cuckoo Hashing

Fotakis, Pagh, Sanders und Spirakis stellen 2003 *d-äres Cuckoo Hashing* vor, das mit $m = (1+\varepsilon)n$ Zellen auskommt, dafür aber für die Suche d wahlfreie Zugriffe in das Feld T benötigt für ein $d = O(-\log \varepsilon)$ [FPSS03]. Bei diesem Verfahren werden d Hashfunktionen $h_i \colon U \to [m/d]$, $i \in [d]$, benutzt und das Feld T in d Teilfelder $T_0, \ldots, T_{d-1}$ gleicher Größe eingeteilt.[1]

[1] T ist stets so gewählt, dass die Größe von T durch d teilbar ist.



Der neu einzufügende Schlüssel wird bei $y_1 = h_1(x_1)$ in T_1 eingefügt. Der Schlüssel x_1, der vorher in dieser Zelle stand, wird bei $y_2 = h_2(x_1)$ in T_2 eingefügt. Der Schlüssel x_2, der vorher in y_2 stand, wird bei $y_3 = h_1(x_2)$ in T_1 eingefügt. Weil $T_1[y_3]$ vorher frei war, ist die Einfügeoperation abgeschlossen.

Abbildung 3.1: Einfügen eines Schlüssels beim Cuckoo Hashing

Der Schlüssel x wird dabei in einer der Zellen $T_i[h_i(x)]$ für ein $i \in [d]$ gespeichert.

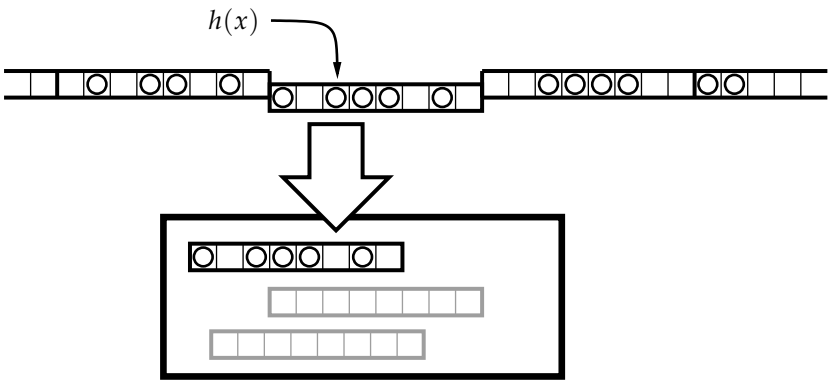
Um das Einfügen eines neuen Schlüssels x zu beschreiben, betrachten wir einen ungerichteten bipartiten Graphen $G = (S, [m], E)$ mit $E = \{\{x', im/d + h_i(x')\} \mid x' \in S, i \in [d]\}$. Die linke Seite des Graphen stellt also die Menge der bereits gespeicherten Schlüssel S dar, und die rechte Seite stellt die Menge der Zellen in T dar, wobei die Zelle j der Tabelle i dem Knoten $y = im/d + j$ auf der rechten Seite entspricht.

Um einen neuen Schlüssel x einzufügen, wird zunächst der Knoten x auf der linken Seite und die Kantenmenge $\{x, im/d + h_i(x)\}$ für $i \in [d]$ auf der rechten Seite zu G hinzugefügt. Anschließend wird mit Breitensuche ein kürzester Pfad von x zu einem Knoten $j \in [m]$ auf der rechten Seite, der einer freien Zelle entspricht, gesucht. Entlang dieses Pfades werden die Schlüssel in den belegten Zellen jeweils in die nächste Zelle in Richtung j verschoben und x wird in der so entstehenden freien Zelle gespeichert.

3.1.4 Bemerkungen

Es zeigt sich, dass Lineares Sondieren dann schnell ist, wenn man auf Löschoperationen verzichtet und ε genügend groß wählt; eine gute Wahl ist zum Beispiel $\varepsilon \geq 0.1$. Dann erwartet man, dass man bei einer positiven Suche im Durchschnitt höchstens 5 Zellen und bei einer negativen Suche 50 Zellen besuchen muss (siehe Abschnitt 3.1.1).

Am Ende dieses Kapitels führen wir mit den verschiedenen Verfahren Experimente durch. Vergleicht man dabei die gemessenen Zeiten von Linearem Sondieren und d-ärem Cuckoo Hashing miteinander, stellt man fest, dass Lineares Sondieren dem d-ärem Cuckoo Hashing in dieser Situation überlegen ist.



Wenn die Zelle $z = h(x)$ gelesen werden soll, wird zunächst der gesamte Block, in dem z liegt, in den Cache geladen. Werden anschließend die auf z folgenden Zellen gelesen, liegen diese schon im Cache.

Abbildung 3.2: Illustration eines Caches

Eine wichtige Ursache für diese Beobachtung ist die mehrschichtige Speicherarchitektur moderner Computersysteme und die Verwendung von Caches [Prz98]: Zugriffe auf Speicherbereiche, die im Cache lagern, sind viel schneller als Zugriffe auf Bereiche, die sich nicht im Cache befinden. Greift man auf eine Speicherzelle zu, die sich nicht im Cache befindet, spricht man von einem Cache-Fehler (*cache miss*, *cache fault*).

Obendrein wird, wenn bei einem Zugriff auf eine Speicherzelle ein Cache-Fehler auftritt, nicht nur die gewünschte Zelle in den Cache geladen, sondern gleich ein ganzer Block zusammenhängender Zellen. Wie ein solcher Block genau aussieht, hängt von der konkreten Architektur des Systems ab. Aber in der Regel beginnen diese Blöcke an durch 2^t teilbaren Adressen. Ist beispielsweise $t = 3$ und hat die gewünschte Zelle die Adresse 1027, dann werden die Zellen mit den Adressen $1024, \ldots, 1031$ in den Cache geladen.

Weiterhin unterstützen viele Cache-Systeme ein vorausschauendes Laden von Speicherzellen (*prefetch*). Durchsucht man zum Beispiel die Zellen $1027, 1028, \ldots$, lädt der Prozessor während der Suche im Block mit den Adressen $1024, \ldots, 1031$ bereits den nächsten Block mit den Adressen $1032, \ldots, 1039$.

Lineares Sondieren passt sehr gut in dieses Konzept, während d-äres Cuckoo Hashing mit den willkürlichen Speicherzugriffen keine Vorteile aus dieser Architektur ziehen kann.

Wie stark ein Algorithmus durch Optimierung der Speicherzugriffe profitieren kann, ist zum Beispiel in [San99] und [BKS00] untersucht.

3.2 Varianten des Cuckoo Hashing

Betrachtet man die bisher genannten Verfahren, drängt sich die Idee auf, das Cuckoo-Prinzip mit dem Linearen Sondieren zu verbinden: Anstatt für einen Schlüssel x nur die beiden Zellen $h_1(x)$ und $h_2(x)$ als mögliche Speicherorte vorzusehen, wollen wir für x zwei jeweils zusammenhängende Bereiche von Zellen, die durch $h_1(x)$ und $h_2(x)$ identifiziert werden, vorsehen. Die Länge der Bereiche soll d, eine Konstante, sein.

Wir werden zeigen, dass man mit einem Feld T der Größe $m = (1+\varepsilon)n$ und zwei zufällig gewählten Hashfunktionen h_1 und h_2 auskommt, wobei $d = O(\log(1/\varepsilon))$ wie beim d-ären Cuckoo Hashing ist. Dabei sind h_1, h_2 zwei Funktionen, so dass die Werte $h_1(x)$ für alle $x \in S$ und auch die Werte $h_2(x)$ für alle $x \in S$ unabhängig voneinander sind (siehe Bemerkung 1 auf Seite 6).

Zunächst wollen wir den Begriff *Zellen, in denen x gespeichert werden kann* definieren:

Definition 7. Sei $d \in \mathbb{N}$ eine Konstante. Die Menge $\Gamma(x)$ der Zellen, in denen ein Schlüssel x gespeichert werden kann, ist

$$\Gamma(x) := \{h_1(x), \ldots, h_1(x) \oplus (d-1), h_2(x), \ldots, h_2(x) \oplus (d-1)\} \qquad (3.2)$$

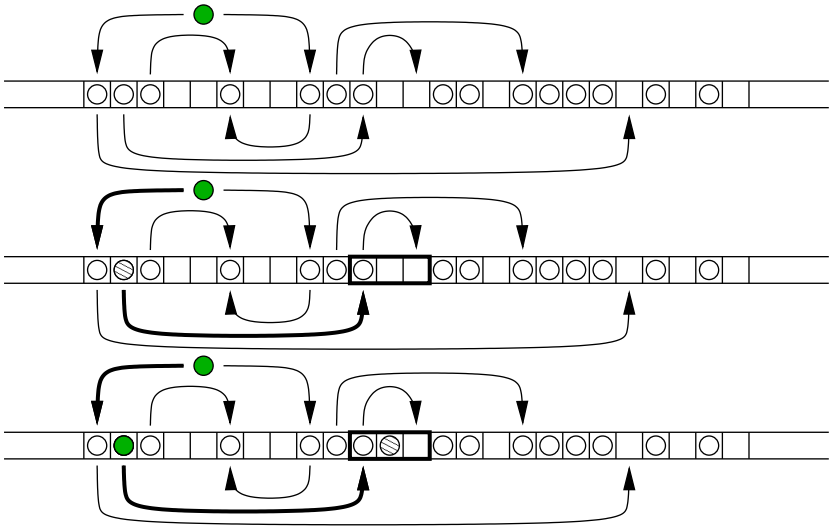
Dabei seien mit den Operationszeichen $\oplus$ und $\ominus$ die Addition und Subtraktion modulo der Größe von T bezeichnet.

In diesem Kapitel werden wir zwei Verfahren angeben, die einen Schlüssel x in einer der Zellen aus $\Gamma(x)$ speichern.

3.2.1 Cuckoo Hashing mit begrenzter Sondierungsweite

Wir wählen $m = (1+\varepsilon)n$. Über die Wahl von d werden wir uns in der Analyse kümmern. Für den Moment sei d eine feste Zahl. Um einen neuen Schlüssel x einzufügen, wird zunächst die Menge $Z = \Gamma(x)$ der Zellen, die man mit $h_1(x)$ und $h_2(x)$ erreichen kann, untersucht. Finden wir in Z eine freie Zelle, fügen wir x dort ein. Finden wir keine freie Zelle, betrachten wir die Nachbarmenge von Z. Die Nachbarmenge finden wir, indem wir für alle Schlüssel x, die in den Zellen Z gespeichert sind, deren Hashwerte berechnen. Die so erhaltenen Hashwerte bestimmen die *Nachbarmenge* von Z.

Unter diesen Nachbarn wird wieder eine freie Zelle gesucht und so weiter. Das Verfahren bricht spätestens dann ab, wenn keine neu zu untersuchenden Zellen hinzukommen.



Sei $d = 3$. Für den neu einzufügenden Schlüssel ist sowohl $h_1(x)$ als auch $h_2(x)$ belegt. Deswegen werden alle Nachbarn der Schlüssel, die in $h_1(x), h_1(x)+1, h_1(x)+2, h_2(x), h_2(x)+1$ und $h_2(x)+2$ gespeichert sind, betrachtet (oberes Bild). Unter diesen Zellen gibt es noch freie Bereiche. Einer dieser Bereiche wird gewählt (dick umrandeter Bereich im mittleren Bild) und der Pfad, der vom einzufügenden Schlüssel in diesen Bereich führt, betrachtet (dick markiert im mittleren Bild). Entlang dieses Pfades wird der schraffierte Schlüssel verschoben und der neue Schlüssel wird in der frei gewordenen Zelle gespeichert (unteres Bild).

Abbildung 3.3: Einfügen eines Schlüssels beim Cuckoo Hashing mit begrenzter Sondierungsweite d

Definition 8. Die Nachbarmenge eines Schlüssel x bei Verwendung fester Blöcke ist

$$\Gamma_B(x) := \{h_1(x), h_2(x)\} \tag{3.3}$$

Der Nachbar einer belegten Zelle $z \in y$ bei Verwendung fester Blöcke ist

$$\zeta_B(z) := \begin{cases} h_2(T[z]), & \text{falls } h_1(T[z]) = y, \\ h_1(T[z]) & \text{sonst.} \end{cases} \tag{3.4}$$

$\zeta_B(z)$ heißt auch *alternativer Block* von z.

Man fragt sich jetzt vielleicht, warum man für eine Zelle z und den darin gespeicherten Schlüssel $x' = T[z]$ erneut alle Zellen bei $h_1(x')$ und $h_2(x')$ betrachtet, obwohl es auf den ersten Blick ausreichen würde, die Zellen um $h_{3-i}(x')$ zu betrachten, wenn x' mit h_i in y gespeichert wurde. Aber man kann sich folgendes Szenario vorstellen: Irgendwann wurde

```
CUCKOO-CACHE-INSERT (x)
1   Z := Γ(x)
2   if ∃z ∈ Z: T[z] = x then return endif
3   Q = CREATE_EMPTY_QUEUE
4   for z ∈ Z do
5     visited(z) := true; ENQUEUE(Q, z)
6     parent(z) := nil
7   endfor
8   while not IS_EMPTY(Q) do
9     z := DEQUEUE(Q)
10    if T[z] = ⊥ then
11      while parent(z) ≠ nil do
12        T[z] := T[parent(z)]; z := parent(z)
13      endwhile
14      T[z] := x
15      return
16    else
17      for z' ∈ Γ(T[z]) do
18        if not visited(z') then
19          parent(z') := z
20          visited(z') := true; ENQUEUE(Q, z')
21        endif
22      endfor
23    endif
24  endwhile
25  error „x konnte nicht eingefügt werden.“
```

```
CUCKOO-CACHE-BLOCK-INSERT (x)
1   Y := Γ_B(x)
2   if ∃y ∈ Y: ∃z ∈ y: T[z] = x then return endif
3   Q = CREATE_EMPTY_QUEUE
4   for y ∈ Y do
5     visited(y) := true; ENQUEUE(Q, y)
6     for z ∈ y do parent(z) := nil endfor
7   endfor
8   while not IS_EMPTY(Q) do
9     y := DEQUEUE(Q)
10    if ∃z ∈ y: T[z] = ⊥ then
11      while parent(z) ≠ nil do
12        T[z] := T[parent(z)]; z := parent(z)
13      endwhile
14      T[z] := x
15      return
16    else
17      for y' ∈ {ζ_B(z) | z ∈ y} do
18        if not visited(y') then
19          for z' ∈ y' do parent(z') := z endfor
20          visited(y') := true; ENQUEUE(Q, y')
21        endif
22      endfor
23    endif
24  endwhile
25  error „x konnte nicht eingefügt werden.“
```

Algorithmus 3.2: Einfügen eines Schlüssels beim Cuckoo Hashing mit festen Blöcken und mit begrenzter Sondierungsweite

der Schlüssel x' in der Zelle $h_1(x') + 2$ gespeichert, weil die Zellen $h_1(x')$ und $h_1(x') + 1$ bereits belegt waren. Werden nun Schlüssel in $h_1(x')$ und $h_1(x') + 1$ gelöscht, lohnt es sich, als Alternativen zur aktuellen Position von x' auch die nun freien Zellen $h_1(x')$ und $h_1(x') + 1$ zu betrachten.

Das Suchen einer freien Zelle kann durch Breitensuche in einem gerichteten Graphen G beschrieben werden, dessen Knoten die Zellen von T sind und dessen Kantenmenge E wie folgt bestimmt ist:

$$E = \{(z, z') \mid \exists x \in S\colon T[z] = x \wedge z, z' \in \Gamma(x)\}.$$

Dabei startet die Breitensuche mit den Knoten $\Gamma(x)$. Hat man eine freie Zelle z gefunden, verschiebt man die Schlüssel auf dem Pfad, der von $\Gamma(x)$ zu z führt, in Richtung z. Damit gilt für jeden verschobenen Schlüssel x', dass er immer noch in einer der Zellen aus $\Gamma(x')$ gespeichert ist. In der so frei gewordenen Zelle speichern wir x.

Dieses Verfahren ist im Algorithmus CUCKOO-CACHE-INSERT (Algorithmus 3.2 auf der vorherigen Seite) implementiert.[2] In der Abbildung 3.3 ist dessen Arbeitsweise illustriert.

Der Suchalgorithmus ist einfach: Es reicht, alle Zellen in $\Gamma(x)$ nach dem Vorkommen von x zu durchsuchen. Dies ist im Algorithmus CUCKOO-CACHE-LOOKUP (Algorithmus 3.3 auf Seite 63) implementiert.

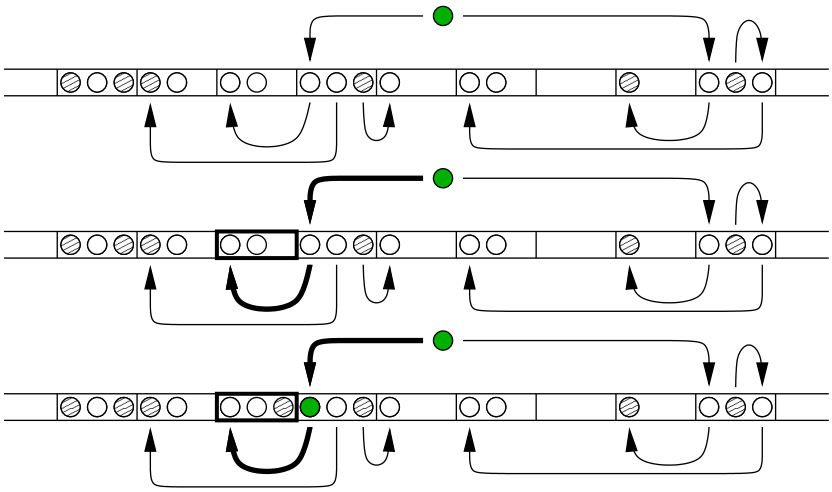
3.2.2 Cuckoo Hashing mit festen Blöcken

Wir betrachten weiter eine Variante des eben beschriebenen Verfahrens. Dieses liegt der anschließenden Analyse zu Grunde. Wir werden aber für beide Varianten experimentelle Resultate angeben.

Sei $m' = (1 + \varepsilon)n$ durch d teilbar. Wir teilen das Feld T in $m = (1 + \varepsilon)n/d$ Blöcke der Größe d, so dass die ersten d Zellen den Block 0 bilden, die nächsten d Zellen den Block 1 und so weiter. Das heißt, dass der Block j mit der Speicherzelle jd beginnt. Die Hashfunktionen h_1 und h_2 sollen jeden Schlüssel x auf die Blöcke abbilden, in denen x gespeichert werden kann, also $h_1, h_2\colon U \to [m]$.

Wenn eine Zelle z zu einem Block y gehört, wollen wir einfach $z \in y$ schreiben. Falls eine Zelle z in einem der Blöcke einer Menge Y von Blöcken enthalten ist, schreiben wir $z \in Y$. Gibt es in einem Block eine freie Zelle, wollen wir den Block als *frei* bezeichnen, ansonsten als *blockiert*. Zunächst wollen wir die Nachbarschaft eines Schlüssels x, einer Zelle z und eines Blockes y definieren.

[2] Weil die Verfahren auf dem Cuckoo-Prinzip von [PR01] beruhen und gut mit Caches harmonieren, nennen wir diese Algorithmen CUCKOO-CACHE-...



Sei $d = 3$. Schraffierte Kreise bedeuten, dass ein Schlüssel mit h_1 an dieser Stelle gespeichert wurde und helle Kreise, dass der Schlüssel mit h_2 gespeichert wurde. Für den neu einzufügenden Schlüssel x sind $h_1(x)$ und $h_2(x)$ blockiert. Deswegen werden die alternativen Blöcke der Schlüssel, die in den Blöcken $h_1(x)$ und $h_2(x)$ gespeichert sind, betrachtet (oberes Bild). Unter diesen Nachbarblöcken gibt es welche, die noch Platz bieten. Einer dieser freien Blöcke wird gewählt (dick umrandeter Block im mittleren Bild) und der Pfad, der vom einzufügenden Schlüssel in diesen Block führt, betrachtet (dick markiert im mittleren Bild). Entlang dieses Pfades wird ein Schlüssel verschoben und der neue Schlüssel wird in der frei gewordenen Zelle gespeichert (unteres Bild). Die verschobenen Schlüssel ändern ihre Schraffur.

Abbildung 3.4: Einfügen eines Schlüssels beim Cuckoo Hashing mit festen Blöcken

Um nun einen neuen Schlüssel x einzufügen, gehen wir ähnlich wie im letzten Abschnitt vor: Wir untersuchen zunächst, ob x bereits in einem der Blöcke $\Gamma_B(x)$ gespeichert ist. Ansonsten führen wir analog zum Verfahren mit begrenzter Sondierungsweite eine Breitensuche durch, um einen freien Platz zu finden und verschieben die Schlüssel entlang des gefundenen Pfades zum freien Feld.

Das Einfügen eines neuen Schlüssels lässt sich auf die Suche nach einem freien Block in einem gerichteten Graphen zurückführen, dessen Knoten die Blöcke sind und dessen Kantenmenge

$$E = \{(h_r(x'), h_{3-r}(x')) \mid x' \in S \wedge x' \text{ ist im Block } h_r(x') \text{ gespeichert}\}$$

ist, wobei die Suche bei den beiden Knoten $h_1(x)$ und $h_2(x)$ beginnt.

Man beachte, dass wir, im Gegensatz zu vorhin, bei dieser Variante des Verfahrens für eine Zelle z nur den alternativen Block des in $T[z]$ gespeicherten Schlüssels x' betrachten müssen, um eine alternative Zelle für x'

```
CUCKOO-CACHE-LOOKUP (x)
1  for z ∈ Γ(x) do
2     if T[z] = x then return z endif
3  endfor
4  return „x nicht gefunden.“
```

```
CUCKOO-CACHE-BLOCK-LOOKUP (x)
1  for y ∈ Γ_B(x) do
2     for z ∈ y do
3        if T[z] = x then return z endif
4     endfor
5  endfor
6  return „x nicht gefunden.“
```

Algorithmus 3.3: Suchen eines Schlüssels beim Cuckoo Hashing mit begrenzter Sondierungsweite und mit festen Blöcken

zu finden.

Unter dem Abschnitt 3.4 auf Seite 93 ist erläutert, wie man die Funktionen h_1 und h_2 so wählen kann, dass sich $\zeta_B(z)$ leicht berechnen lässt.

Der Algorithmus CUCKOO-CACHE-BLOCK-INSERT zum Einfügen ist unter Algorithmus 3.2 auf Seite 60 zu finden. In Abbildung 3.4 ist der Algorithmus illustriert. Zum Suchen eines Schlüssels x verwenden wir den Algorithmus CUCKOO-CACHE-BLOCK-LOOKUP (Algorithmus 3.3).

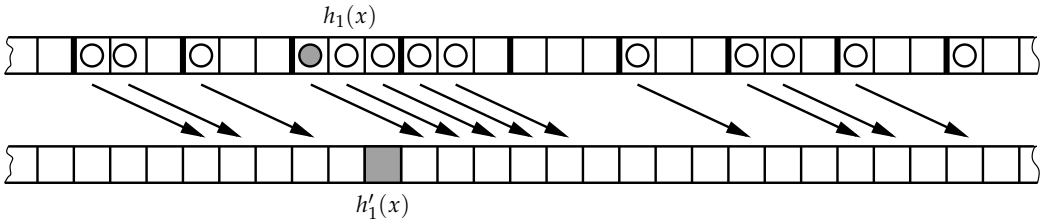
3.2.3 Bemerkungen zu den Varianten

Im nächsten Abschnitt werden wir uns der Analyse des Verfahrens mit festen Blöcken widmen. Wir werden zeigen, dass für $d \geq 1 + \frac{1}{1-\ln 2} \ln \frac{1}{\varepsilon}$ mit hoher Wahrscheinlichkeit n Schlüssel in einem Feld mit $m' = (1+\varepsilon)n$ Plätzen, das heißt in $m = (1+\varepsilon)n/d$ Blöcken der Größe d, abgespeichert werden können (Lemma 7 auf Seite 67). Diese Analyse impliziert sofort eine Schranke für die Sondierungesweite bei der Variante mit fester Sondierungsweite.

Bemerkung 5. Wenn es möglich ist, bei Verwendung fester Blöcke die Schlüssel der Schlüsselmenge S im Feld T, das in $m = (1+\varepsilon)n/d$ Blöcke der Größe d aufgeteilt ist, zu speichern, kann man alle Schlüssel auch bei Verwendung einer begrenzten Sondierungsweite (siehe Abschnitt 3.2.1 auf Seite 58) der Größe $2d$ in T abspeichern.

Beweis. Seien $h'_1, h'_2 \colon U \to [m']$ die beiden Hashfunktionen für das Verfahren mit begrenzter Sondierungsweite. Dann seien $h_1, h_2 \colon U \to [m]$, mit $h_j(x) := \lfloor h'_j(x)/d \rfloor$, $j = 1, 2$, die Funktionen, die einen Schlüssel x auf seine möglichen Blöcke abbildet.

Seien nun die Schlüssel bei Verwendung fester Blöcke der Größe d mit h_1 und h_2 gespeichert. Dann kann man alle Schlüssel, die im Block i gespeichert sind, in den Zellen $T[id \oplus d], \ldots, T[id \oplus (2d-1)]$ speichern (siehe



Alle Schlüssel, die im den Block i gespeichert sind (oben), werden in $T[di \oplus d], \ldots, T[di \oplus (2d-1)]$ gespeichert (unten). Dabei ist $d = 3$. Sei der dunkel markierte Schlüssel x bei Verwendung fester Blöcke im Block $y = h'_1(x)$ gespeichert. Bei Verwendung einer festen Sondierungsweite $2d$ ist er in einer der Zellen $T[dy \oplus d], \ldots, T[dy \oplus (2d-1)]$ gespeichert und hat eine Entfernung von weniger als $2d$ zur Zelle $h_1(x)$ (grau hinterlegt).

Abbildung 3.5: Verwendung fester Blöcke versus Verwendung fester Sondierungsweite

Abbildung 3.5). Damit gilt für jeden Schlüssel $x \in S$, dass er bei Verwendung einer begrenzten Sondierungsweite von $2d$ in einer der Zellen aus $\Gamma(x)$ (siehe Definition 7 auf Seite 58) gespeichert ist. □

Bemerkung 6. In den Experimenten zeigt sich, dass das Verfahren mit begrenzter Sondierungsweite mit dem gleichen oder sogar einem kleineren d auskommt, als das Verfahren mit festen Blöcken.

Nach der Analyse für den statischen Fall zeigen wir auch, dass für $d \geq 90.1 \cdot \ln(1/\varepsilon)$ die erwartete Einfügezeit für einen neuen Schlüssel konstant ist (Lemma 15 auf Seite 88).

Die Anzahl der zu untersuchenden Zellen bei einer Suche nach einem Schlüssel in einer der beiden Wörterbuchvarianten unterscheidet sich vom d-ären Cuckoo Hashing lediglich um einen konstanten Faktor. Vergleicht man die Zeiten für das Suchen und das Einfügen eines neuen Schlüssel mit den entsprechenden Zeiten beim Uniformen Sondieren (*uniform probing*) [Knu82, Abschnitt 6.4], stellt man folgendes fest:

- Beim Uniformen Sondieren ist die mittlere erwartete Suchzeit bei einer erfolgreichen Suche $O(\log(1/\varepsilon))$ und die erwartete Suchzeit bei einer erfolglosen Suche $O(1/\varepsilon)$.
- Bei unseren beiden Wörterbuchvarianten wie auch beim d-ären Cuckoo Hashing ist die Suchzeit im schlechtesten Fall $O(\log(1/\varepsilon))$.
- Beim Uniformen Sondieren entspricht das Einfügen eines neuen Schlüssels einer erfolglosen Suche. Das heißt, die erwartete

Einfügezeit ist $O(1/\varepsilon)$. Bei der Wörterbuchvariante mit festen Blockgrenzen kann man für erwartete Zeit zum Einfügen eines neuen Schlüssel eine obere Schranke von $(1/\varepsilon)^{O(\log d)}$ beweisen. Beim d-ären Cuckoo Hashing ist diese Schranke ebenfalls $(1/\varepsilon)^{O(\log d)}$. Die der Analyse unserer Verfahren folgenden experimentellen Resultate lassen aber vermuten, dass die erwartete Einfügezeit bei den neuen Wörterbüchern und beim d-ären Cuckoo Hashing viel kleiner sind.

3.3 Analyse bei Verwendung fester Blöcke

In diesem Abschnitt wollen wir untersuchen, wann es möglich ist, n Schlüssel in einem Feld T der Größe $m' = (1+\varepsilon)n$ zu speichern, wenn wir das Verfahren mit festen Blöcken verwenden. Das heißt, wir verwenden $m = m'/d$ Blöcke wie eben beschrieben. Insbesondere interessiert uns die Größe von d, um alle Schlüssel einer Menge S zu speichern und um eine erwartete konstante Einfügezeit zu erhalten.

Ähnliche Untersuchungen wurden in jüngster Vergangenheit schon vorgenommen.

- In [ABKU00] wird gezeigt, dass man mit hoher Wahrscheinlichkeit n Bälle in n Behältern unterbringen kann, so dass in jedem Behälter höchstens $\frac{\ln\ln n}{\ln 2} + O(1)$ Bälle liegen. Dabei werden für jeden Ball zwei Behälter zufällig gewählt, und nachdem ein Ball platziert worden ist, wird seine Position nicht mehr verändert. Wenn es erlaubt ist, die Position bereits platzierter Bälle zu verändern und $n \leq 1.67m$ gilt, kann man die Bälle so unterbringen, dass mit hoher Wahrscheinlichkeit kein Behälter mehr als zwei Bälle enthält. Überträgt man dies auf unser Verfahren, folgt für $d = 2$ und wegen $m = (1+\varepsilon)n/d$ sofort $d/(1+\varepsilon) < 1.67$ und schließlich $\varepsilon \geq 0.2$.

- In [BCSV00] wird gezeigt, dass man n Bälle in m Behältern so unterbringen kann, dass jeder Behälter höchstens $\frac{n}{m} + O(\ln\ln m)$ Bälle enthält. Dabei werden für jeden Ball $d \geq 2$ Behälter zufällig gewählt.

- In [CRS03] wurde gezeigt, dass eine perfekte Auslastung der Behälter ($n = dm$, das heißt $\varepsilon = 0$) mit hoher Wahrscheinlichkeit nicht zu erreichen ist, falls $d < \gamma_1 \ln n$ für eine passende Konstante γ_1, während eine perfekte Auslastung möglich ist, falls $d > \gamma_2 \ln n$ für eine weitere Konstante $\gamma_2 > \gamma_1$. Außerdem wurde in [CRS03] ein randomisierter Algorithmus mit polynomieller Laufzeit in n zum Ausbalancieren angegeben und analysiert.

- In [SEK00, San01] wird folgendes Problem untersucht: Gegeben sind N Anforderungen, jeweils einen Datensatz von einer von D parallel betriebenen Festplatte zu laden. Dabei gibt es für jeden Datensatz zwei zufällig gewählte Platten, auf denen er gespeichert ist. In der genannten Arbeit ist gezeigt, dass mit Wahrscheinlichkeit $1 - O(1/D)^{b+1}$ höchstens $b+1$ Anforderungen an jede Platte gesendet werden müssen, wobei $b = \lceil N/D \rceil$.

  Das in den genannten Arbeiten untersuchte Problem entspricht exakt unserem: N ist die Anzahl der Schlüssel n, D ist die Anzahl der Blöcke m und die Anzahl der Anforderungen, die an jede Platte geschickt werden, ist die Zahl der Schlüssel, die in jedem Block gespeichert sind, das heißt für unser d gilt $d = 1 + b$. Wir wollen davon ausgehen, dass D durch N teilbar ist. Wir wenden nun das Resultat dieser Arbeiten an: Damit $N = n$ Schlüssel in $(1+\varepsilon)N$ Zellen gespeichert werden können, muss die Ungleichung $D(1+b) \leq (1+\varepsilon)N$ erfüllt sein. Wenn wir $b = N/D$ einsetzen, sehen wir, dass das genau dann der Fall ist, wenn $1 + b \leq (1+\varepsilon)b$ gilt. Dies wiederum ist genau dann der Fall, wenn $b \geq 1/\varepsilon$ gilt, das heißt $d \geq 1 + 1/\varepsilon$.

  Sieht man sich den Beweis der Aussage in den genannten Arbeiten an, stellt man fest, dass ein ähnlicher Ansatz wie in der im nächsten Abschnitt folgenden Analyse verwendet wird. Insbesondere die Ungleichung (3.12) wird dort hergeleitet, allerdings lediglich experimentell – für $d = 2, \ldots, 9$ wurden die kleinsten möglichen ε per Kurvenplot ermittelt – untersucht. Diese Beziehung wird in der vorliegenden Arbeit analytisch untersucht. Dadurch erhalten wir eine präzisere Formel für d als in den genannten Arbeiten.

Nachdem wir geklärt haben, unter welchen Bedingungen wir überhaupt eine Schlüsselmenge S speichern können, untersuchen wir den Algorithmus CUCKOO-CACHE-BLOCK-INSERT hinsichtlich seiner erwarteten Laufzeit. Die Strategie der Beweisführung ist der in [FPSS03] sehr ähnlich.

3.3.1 Analyse für den statischen Fall

Wir untersuchen die Frage, wie groß d gewählt werden muss, damit eine Menge S von Schlüsseln in $m = (1+\varepsilon)n/d$ Blöcken der Größe d gespeichert werden kann.

Eine Schlüsselmenge S kann genau dann in $m = (1+\varepsilon)n/d$ Blöcken der Größe d gespeichert werden, wenn für alle $X \subseteq S$ gilt: $d|\Gamma_B(X)| \geq |X|$. Deswegen schätzen wir die Wahrscheinlichkeit für das Ereignis, dass es ein

$X \subseteq S$ mit $d|\Gamma_B(X)| < |X|$ gibt, ab. Dieses Ereignis nennen wir F. Wir werden zeigen, dass $\mathbf{Prob}(F)$ sehr klein ist.

Lemma 7. *Sei $\varepsilon \leq 0.25$ und $d > 1 + \frac{\ln(1/\varepsilon)}{1-\ln 2}$. Dann gilt:* $\mathbf{Prob}(F) \in O(m^{1-d})$.

Beweis. Sei $F(j)$ das Ereignis, dass es eine Menge Y von j Blöcken gibt, für die es eine Menge $X \subseteq S$ von dj Schlüsseln mit $\Gamma_B(X) \subset Y$ gibt. Das Ereignis $F(j)$ kann nur eintreten, falls $dj \leq n$ gilt. Das heißt, es gilt $1 \leq j \leq n/d = m/(1+\varepsilon)$, und damit ist

$$F = \bigcup_{j=1}^{m/(1+\varepsilon)} F(j) \tag{3.5}$$

und

$$\mathbf{Prob}(F) \leq \sum_{j=1}^{m/(1+\varepsilon)} \mathbf{Prob}(F(j)). \tag{3.6}$$

Um $\mathbf{Prob}(F(j))$ abzuschätzen, betrachten wir für ein festes Y der Größe j folgendes Experiment: Wir ziehen der Reihe nach alle Schlüssel $x \in S$ aus einer Urne und notieren für jeden Schlüssel die Zufallsvariable I_x, wobei

$$I_x := \begin{cases} 1, & \text{falls } h_1(x), h_2(x) \in Y \\ 0 & \text{sonst.} \end{cases} \tag{3.7}$$

Sei außerdem

$$I = \sum_{x \in S} I_x. \tag{3.8}$$

Das Ereignis $F(j)$ tritt nur dann ein, wenn $I > dj$ gilt. Die Wahrscheinlichkeit $\mathbf{Prob}(I > dj)$ können wir mit der Chernoff-Hoeffding-Schranke abschätzen.

Zunächst stellen wir fest, dass die Variablen I_x unabhängig voneinander sind, also gilt für jedes j:

$$\mathbf{Prob}(I_x = 1) = (j/m)^2 \tag{3.9}$$

Damit bekommen wir den Erwartungswert von I:

$$\mathbf{E}(I) = \sum_{x \in S} E(I_x) = \sum_{x \in S} \mathbf{Prob}(I_x) = n\frac{j^2}{m^2}. \tag{3.10}$$

Die Chernoff-Hoeffding-Schranke liefert damit

$$\begin{aligned}\mathbf{Prob}(I > jd) &\leq \left(\frac{E(I)}{jd}\right)^{jd} \left(\frac{n - E(I)}{n - jd}\right)^{n-jd} \\ &= \left(\frac{nj^2}{m^2 jd}\right)^{jd} \left(\frac{n(m^2 - j^2)}{m^2(n - jd)}\right)^{n-jd} \qquad (3.11)\end{aligned}$$

Um schließlich $F(j)$ abzuschätzen, bedenken wir, dass es $\binom{m}{j}$ verschiedene Mengen Y der Größe j gibt. Somit bekommen wir

$$\mathbf{Prob}(F(j)) \leq \binom{m}{j} \left(\frac{nj^2}{m^2 jd}\right)^{jd} \left(\frac{n(m^2 - j^2)}{m^2(n - jd)}\right)^{n-jd}. \qquad (3.12)$$

Mit der Ungleichung 1.5 und der Beziehung

$$n = \frac{dm}{1 + \varepsilon} \qquad (3.13)$$

formen wir den letzten Ausdruck um und bekommen:

$$\begin{aligned}\mathbf{Prob}(F(j)) &\leq \frac{m^m}{j^j (m - j)^{m-j}} \left(\frac{j}{(1 + \varepsilon)m}\right)^{jd} \left(\frac{d(m^2 - j^2)}{(1 + \varepsilon)m(n - jd)}\right)^{dm/(1+\varepsilon)-jd} \\ &= \frac{m^m}{j^j (m - j)^{m-j}} \left(\frac{j}{(1 + \varepsilon)m}\right)^{jd} \left(\frac{m^2 - j^2}{(1 + \varepsilon)m(m/(1 + \varepsilon) - j)}\right)^{dm/(1+\varepsilon)-jd} \\ &= (1 + \varepsilon)^{-jd} m^{\frac{m(1+\varepsilon-d)}{1+\varepsilon}} j^{j(d-1)} (m - j)^{j-m} \left(\frac{m^2 - j^2}{m - j(1 + \varepsilon)}\right)^{d\frac{m-j(1+\varepsilon)}{1+\varepsilon}}. \qquad (3.14)\end{aligned}$$

Den Ausdruck auf der rechten Seite von Ungleichung (3.14) wollen wir von nun an einfach f nennen. Wir untersuchen $\mathbf{Prob}(F(j))$ in verschiedenen Bereichen:

Fall 1: $j = 1$**.** Mit Ungleichung (3.12) und Gleichung (3.13) erhalten wir:

$$\mathbf{Prob}(F(1)) \leq m \left(\frac{1}{(1 + \varepsilon)m}\right)^{d} \left(\frac{n - n/m^2}{n - d}\right)^{n-d} \qquad (3.15)$$

Mit

$$\frac{n - n/m^2}{n - d} = 1 + \frac{d - n/m^2}{n - d} \leq e^{\frac{d-n/m^2}{n-d}} \qquad (3.16)$$

erhalten wir

$$\mathbf{Prob}(F(1)) \leq m \left(\frac{1}{(1+\varepsilon)m}\right)^d e^{d-n/m^2} < \left(\frac{e}{1+\varepsilon}\right)^d \frac{1}{m^{d-1}} = O\left(\frac{1}{m^{d-1}}\right). \quad (3.17)$$

Fall 2: $2 \leq j < e^{-4}m$**.** Wir zeigen zunächst, dass wir $\varepsilon = 0$ setzen dürfen. Dazu bilden wir

$$\frac{\partial \ln f}{\partial \varepsilon} = -d(1+\varepsilon)^{-2} m \ln \frac{1-(j/m)^2}{1-j(1+\varepsilon)/m}. \quad (3.18)$$

Aus $j/m < 1$ folgt $1-(j/m)^2 > 1-(1+\varepsilon)j/m$. Damit sehen wir, dass $\frac{\partial \ln f}{\partial \varepsilon} < 0$. Deshalb können wir in (3.14) $\varepsilon = 0$ setzen:

$$\begin{aligned}
\mathbf{Prob}(F(j)) &< m^{m(1-d)} j^{j(d-1)} (m-j)^{j-m} (m+j)^{d(m-j)} \\
&= m^{m(1-d)} j^{j(d-1)} m^{j-m} \left(1-\frac{j}{m}\right)^{j-m} m^{d(m-j)} \left(1+\frac{j}{m}\right)^{d(m-j)} \\
&< m^{j(1-d)} j^{j(d-1)} e^{\frac{j(m-j)}{m}} e^{\frac{dj(m-j)}{m}} = \left(\left(\frac{j}{m}\right)^{d-1} e^{(d+1)(1-j/m)}\right)^j \\
&< \left(\left(\frac{j}{m}\right)^{d-1} e^{d+1}\right)^j \quad (3.19)
\end{aligned}$$

Die Ableitung des letzten Ausdrucks nach j ist

$$\left(e^{d+1}\left(\frac{j}{m}\right)^{d-1}\right)^j \cdot \left(\ln\left(e^{d+1}\left(\frac{j}{m}\right)^{d-1}\right) + d - 1\right). \quad (3.20)$$

Aus $j/m \leq e^{-4}$ folgt

$$\left(\frac{j}{m}\right)^{d-1} < e^{-2d} \quad \Rightarrow \quad e^{d+1}\left(\frac{j}{m}\right)^{d-1} < e^{1-d} \quad (3.21)$$

und damit, dass der Ausdruck (3.20) negativ ist. Deswegen folgt für $2 \leq j < e^{-4}m$:

$$\mathbf{Prob}(F(j)) \leq \mathbf{Prob}(F(2)) < e^{2d+2} \cdot 2^{2d-2} \cdot \frac{1}{m^{2d-2}} = O\left(\frac{1}{m^{2d-2}}\right). \quad (3.22)$$

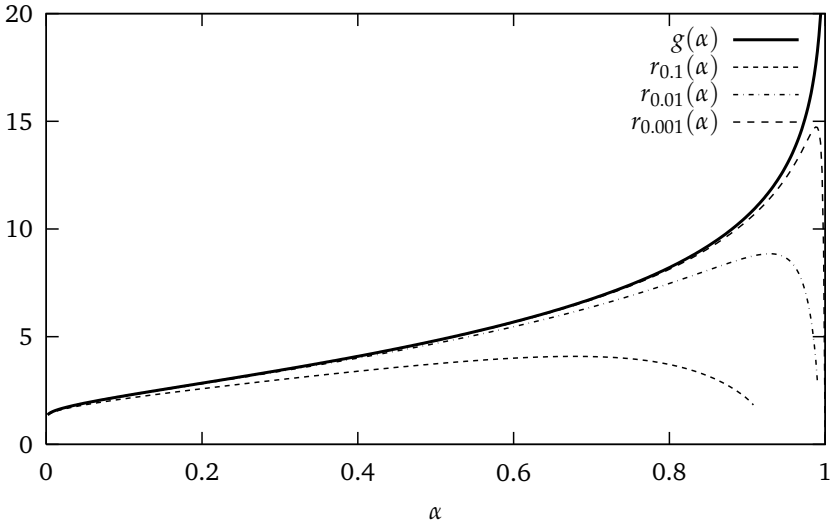


Abbildung 3.6: Verlauf der Kurven g und r_ε für verschiedene ε

Fall 3: $e^{-4}m \leq j \leq (1-2\varepsilon)m$**.** Setzen wir in Ungleichung (3.14) in der rechten Seite $\alpha = j/m$ und logarithmieren, erhalten wir $\ln(\mathbf{Prob}(F(j))) \leq mR_\varepsilon(\alpha)$ mit

$$R_\varepsilon(\alpha) := -\alpha\ln\alpha - (1-\alpha)\ln(1-\alpha) + \alpha d(\ln\alpha - \ln(1+\varepsilon)) + \frac{d(1-\alpha(1+\varepsilon))(\ln(1-\alpha^2) - \ln(1-\alpha(1+\varepsilon)))}{1+\varepsilon}. \quad (3.23)$$

Wir wollen d so wählen, dass $R_\varepsilon(\alpha) < 0$ gilt. Dann folgt $\mathbf{Prob}(F(j)) \leq c^m$ für eine Konstante $c < 1$. Um ein geeignetes d zu finden, stellen $R_\varepsilon(\alpha)$ nach d um und erhalten:

$$d > \frac{\alpha\ln\alpha + (1-\alpha)\ln(1-\alpha)}{\alpha(\ln\alpha - \ln(1+\varepsilon)) + \frac{(1-\alpha(1+\varepsilon))(\ln(1-\alpha^2)-\ln(1-\alpha(1+\varepsilon)))}{1+\varepsilon}} =: r_\varepsilon(\alpha) \quad (3.24)$$

Den Bruch werden wir abschätzen. Wir zeigen zuerst, dass

$$g(\alpha) := \frac{\alpha\ln\alpha + (1-\alpha)\ln(1-\alpha)}{\alpha\ln\alpha + (1-\alpha)\ln(1+\alpha)} \quad (3.25)$$

eine obere Schranke für r_ε ist (siehe Abbildung 3.6).

Wir haben es hier mit zwei Brüchen A/B und A/B' zu tun, deren beider Zähler und Nenner negativ sind. Damit gilt: $A/B' \geq A/B \Leftrightarrow$

$1/B' \leq 1/B \Leftrightarrow B' \geq B$. Wenn wir nun zeigen, dass

$$\begin{aligned}
&0 \leq \alpha \ln \alpha + (1-\alpha)\ln(1+\alpha) - \\
&\left(\alpha(\ln\alpha - \ln(1+\varepsilon)) + \frac{(1-\alpha(1+\varepsilon))(\ln(1-\alpha^2) - \ln(1-\alpha(1+\varepsilon)))}{1+\varepsilon}\right) \\
&= (1-\alpha)\ln(1+\alpha) + \alpha\ln(1+\varepsilon) - \left(\frac{1}{1+\varepsilon} - \alpha\right)\ln\frac{1-\alpha^2}{1-\alpha(1+\varepsilon)},
\end{aligned} \tag{3.26}$$

dann ist der Ausdruck g eine obere Schranke für r_ε. Das ist nicht schwer: Leitet man den letzten Ausdruck nach ε ab, erhält man

$$\frac{\ln(1-\alpha^2) - \ln(1-\alpha(1+\varepsilon))}{(1+\varepsilon)^2} = \frac{\ln\frac{1-\alpha^2}{1-\alpha(1+\varepsilon)}}{(1+\varepsilon)^2} > 0. \tag{3.27}$$

Um (3.26) zu zeigen, reicht es somit, dort $\varepsilon = 0$ zu setzen. Wir erhalten dann:

$$\begin{aligned}
&(1-\alpha)\ln(1+\alpha) + \alpha\ln(1+\varepsilon) - \left(\frac{1}{1+\varepsilon} - \alpha\right)\ln\frac{1-\alpha^2}{1-\alpha(1+\varepsilon)} \\
&\geq (1-\alpha)\ln(1+\alpha) - (1-\alpha)\left(\ln(1-\alpha^2) - \ln(1-\alpha)\right) = 0.
\end{aligned} \tag{3.28}$$

Nun schätzen wir $g(\alpha)$ ab. Zunächst sehen wir, dass

$$g(\alpha) = 1 + \frac{\ln(1-\alpha) - \ln(1+\alpha)}{\frac{\alpha}{1-\alpha}\ln\alpha + \ln(1+\alpha)}. \tag{3.29}$$

Wir bilden die jeweils zweite Ableitung des Zählers und des Nenners. Für den Zähler erhalten wir

$$(\ln(1-\alpha) - \ln(1+\alpha))'' = \frac{-4\alpha}{(\alpha-1)^2(\alpha+1)^2} < 0 \tag{3.30}$$

und für den Nenner

$$\left(\frac{\alpha}{1-\alpha}\ln\alpha + \ln(1+\alpha)\right)'' = \frac{-2\alpha(\alpha+1)^2\ln\alpha - (1-\alpha)(5\alpha^2+2\alpha+1)}{(\alpha-1)^3\alpha(\alpha+1)^2}. \tag{3.31}$$

Setzt man im Zähler des letzten Ausdrucks $\alpha := 1-\beta$ und setzt für $\ln(1-\beta)$ die Taylor-Reihe ein, sieht man anhand des Koeffizientenvergleiches, dass der Zähler eine Summe $\sum_{k\geq 3} b_i\beta^i$ mit den Koeffizienten $b_i < 0$, $i \geq 3$ ist. Weil auch der Nenner von (3.31) negativ ist, ist die zweite Ableitung des Nenners von (3.29) positiv.

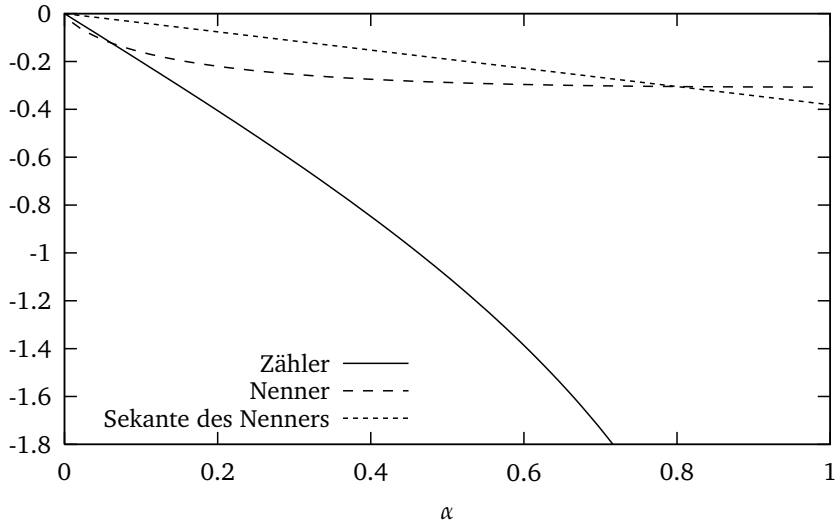


Abbildung 3.7: Illustration von Zähler, Nenner und Sekante des Bruches von Ausdruck (3.29), $\varepsilon = 0.1$

Sowohl Zähler als auch Nenner von (3.29) haben bei $\alpha = 0$ den Wert 0. Ersetzen wir den Nenner durch eine Gerade $A\alpha$ mit $A < 0$, erhalten wir

$$\hat{g}(\alpha) := 1 + \frac{\ln(1-\alpha) - \ln(1+\alpha)}{A\alpha}. \tag{3.32}$$

Die Ableitung dieser Funktion ist

$$\frac{-2\alpha + (1-\alpha^2)\ln\frac{1+\alpha}{1-\alpha}}{A\alpha^2(1-\alpha^2)}. \tag{3.33}$$

Mit

$$\ln\frac{1+\alpha}{1-\alpha} = 2\alpha + \frac{2}{3}\alpha^3 + \frac{2}{5}\alpha^5 + \ldots \tag{3.34}$$

folgt, dass

$$-2\alpha + (1-\alpha^2)\ln\frac{1+\alpha}{1-\alpha} < 0. \tag{3.35}$$

Also ist der Zähler des Ausdrucks (3.33) negativ für $0 < \alpha < 1$. Das heißt, dass wegen $A < 0$ der Ausdruck selber positiv ist. Damit ist $\hat{g}$ monoton wachsend.

Wählen wir für A den Wert

$$\frac{\frac{(1-2\varepsilon)\ln(1-2\varepsilon)}{2\varepsilon} + \ln(2-2\varepsilon)}{1-2\varepsilon}, \tag{3.36}$$

	1	ε	ε^2	ε^3	$\dots$	$\varepsilon^k (k \geq 2)$	$\dots$
$\ln(1-2\varepsilon)$	0	$-\frac{2}{1}$	$-\frac{4}{2}$	$-\frac{8}{3}$	$\dots$	$-\frac{2^k}{k}$	$\dots$
$-2\varepsilon\ln(1-2\varepsilon)$	0	0	$-\frac{8}{2}$	$-\frac{16}{3}$	$\dots$	$-\frac{2^k}{k+1}$	$\dots$
$(1-2\varepsilon)\ln(1-2\varepsilon)$	0	$-\frac{2}{1}$	$\frac{4}{2}$	$\frac{8}{6}$	$\dots$	$\frac{2^k}{(k-1)k}$	$\dots$
$\frac{(1-2\varepsilon)\ln(1-2\varepsilon)}{2\varepsilon}$	-1	$\frac{2}{2}$	$\frac{4}{6}$	$\frac{8}{12}$	$\dots$	$\frac{2^k}{k(k+1)}$	$\dots$
$\ln(1-\varepsilon)$	0	-1	$-\frac{1}{2}$	$-\frac{1}{3}$	$\dots$	$-\frac{1}{k}$	$\dots$
$\frac{(1-2\varepsilon)\ln(1-2\varepsilon)}{2\varepsilon}+1+\ln(1-\varepsilon)$	0	0	$\frac{1}{6}$	$\frac{1}{3}$	$\dots$	$\frac{1}{k}\left(\frac{2^k}{k+1}-1\right)$	$\dots$

Tabelle 3.1: Koeffizienten der Reihe von $\frac{(1-2\varepsilon)\ln(1-2\varepsilon)}{2\varepsilon}+1+\ln(1-\varepsilon)$

dann ist $A\alpha$ eine Sekante der Kurve des Nenners von (3.29) und schneidet dessen Kurve bei $\alpha = (1-2\varepsilon)$. Daraus folgt: $\frac{\alpha}{1-\alpha}\ln\alpha + \ln(1+\alpha) \leq A\alpha$. Für $0 \leq \alpha \leq 1-2\varepsilon$ folgt $\hat{g}(\alpha) \geq g(\alpha)$.[3] An den Stellen $\alpha = 0$ und $\alpha = 1-2\varepsilon$ gilt sogar $\hat{g}(\alpha) = g(\alpha)$. Da $\hat{g}$ monoton wachsend ist, erhalten wir schließlich

$$\hat{g}(\alpha) \leq \hat{g}(1-2\varepsilon) = g(1-2\varepsilon) = 1 + \frac{\ln\varepsilon - \ln(1-\varepsilon)}{\frac{(1-2\varepsilon)\ln(1-2\varepsilon)}{2\varepsilon} + \ln 2 + \ln(1-\varepsilon)}. \tag{3.37}$$

Wir wollen nun zeigen, dass $\hat{g}(1-2\varepsilon) \leq 1 + \frac{-\ln\varepsilon}{1-\ln 2}$ ist. Das ist genau dann der Fall, wenn

$$\left(\frac{(1-2\varepsilon)\ln(1-2\varepsilon)}{2\varepsilon} + 1 + \ln(1-\varepsilon)\right)\ln\varepsilon - (1-\ln 2)\ln(1-\varepsilon) \geq 0. \tag{3.38}$$

Die Reihenentwicklung von $\frac{(1-2\varepsilon)\ln(1-2\varepsilon)}{2\varepsilon}+1+\ln(1-\varepsilon)$ kann man in Tabelle 3.1 ablesen. Betrachtet man die Koeffizienten c_k von ε^k, stellt man fest, dass für $k \geq 2$

$$2c_k - c_{k+1} = \frac{2^{k+1} - (k+2)^2}{k(k+1)(k+2)} \geq 0 \tag{3.39}$$

gilt, also $c_{k+1} \leq 2c_k$. Ist $\varepsilon \leq 0.25$, dann gilt $a_{k+1}\varepsilon^{k+1} \leq \frac{1}{2}a_k\varepsilon^k$ und damit

$$\sum_{k\geq 3} a_k\varepsilon^k \leq a_2\varepsilon^2 = \frac{1}{6}\varepsilon^2. \tag{3.40}$$

[3] Ein bisschen anschaulicher: $A\alpha$ liegt näher an der 0 als $\frac{\alpha}{1-\alpha}\ln\alpha + \ln(1+\alpha)$. Aus diesem Grund gilt die Ungleichung.

Das hilft uns, um Ungleichung (3.38) zu zeigen, denn nun folgt:

$$\left(\frac{(1-2\varepsilon)\ln(1-2\varepsilon)}{2\varepsilon}+1+\ln(1-\varepsilon)\right)\ln\varepsilon-(1-\ln 2)\ln(1-\varepsilon)$$
$$\geq \frac{1}{3}\varepsilon^2\ln\varepsilon-(1-\ln 2)\ln(1-\varepsilon)\geq\frac{1}{3}\varepsilon^2\ln\varepsilon+(1-\ln 2)\varepsilon. \quad (3.41)$$

Die Funktion $\varepsilon\ln\varepsilon$ fällt für $0\leq\varepsilon\leq 0.25$ und erreicht ihr lokales Minimum am rechten Rand. Damit können wir weiter rechnen:

$$\frac{1}{3}\varepsilon^2\ln\varepsilon+(1-\ln 2)\varepsilon>-0.12\varepsilon+0.31\varepsilon>0. \quad (3.42)$$

Wählt man also

$$d\geq 1+\frac{1}{1-\ln 2}\ln\frac{1}{\varepsilon}=1+3.25889\ldots\ln\frac{1}{\varepsilon}, \quad (3.43)$$

dann ist $\mathbf{Prob}(F(j))=O(c^n)$ für eine Konstante $c<1$.

Fall 4: $j\geq(1-2\varepsilon)m$**.** Betrachtet man die erste Ableitung von r_ε (siehe Fall 3, (3.23)) bei $\alpha=1-2\varepsilon$, erhält man

$$\frac{((1-2\varepsilon)\ln(1-2\varepsilon)+2\varepsilon\ln\varepsilon)(2\ln 2+\ln(1-\varepsilon)-\ln(1+2\varepsilon))}{\left((1-\varepsilon-\varepsilon^2)\ln\frac{1+\varepsilon}{1-2\varepsilon}-(2\varepsilon^2+\varepsilon)\ln\frac{4(1-\varepsilon)}{1+2\varepsilon}\right)^2}. \quad (3.44)$$

Wir zeigen, dass der Zähler dieses Bruches für $\varepsilon\leq 0.25$ negativ ist. Dann folgt nämlich, dass der Wert der Ableitung von r_ε bei $\alpha=1-2\varepsilon$ negativ ist.

Um zu zeigen, dass der Zähler negativ ist, betrachten wir dessen zwei Faktoren. Offensichtlich gilt $(1-2\varepsilon)\ln(1-2\varepsilon)+2\varepsilon\ln\varepsilon<0$. Mit der Ungleichung $\ln(1-x)\geq -x/(1-x)$ für $0\leq x<1$ erhalten wir für $\varepsilon\leq 0.25$:

$$2\ln 2+\ln(1-\varepsilon)-\ln(1+2\varepsilon)=2\ln 2+\ln\left(1-\frac{3\varepsilon}{1+2\varepsilon}\right)$$
$$>2\ln 2-\frac{3\varepsilon}{1-\varepsilon}>0. \quad (3.45)$$

Sei u der Nenner von r_ε. Wir bilden zunächst die zweite Ableitung von u:

$$u''=\frac{2\alpha(1+\alpha^2)(2\varepsilon+\varepsilon^2+2)-(1+\varepsilon)(6\alpha^2+1+\alpha^4)}{\alpha(\alpha^2-1)^2(\varepsilon\alpha+\alpha-1)(1+\varepsilon)} \quad (3.46)$$

Der Zähler von u'' ist ein Polynom vierten Grades und kann somit exakt gelöst werden. Dabei findet man wegen $\varepsilon < 0.25$ genau eine Nullstelle in $(0, 1/(1+\varepsilon)]$, nämlich $1+\varepsilon-\sqrt{\varepsilon^2+2\varepsilon}$. Da $\varepsilon < 0.25$, gilt

$$1+\varepsilon-\sqrt{\varepsilon^2+2\varepsilon} < 1-2\varepsilon, \tag{3.47}$$

das heißt u'' hat keine Nullstelle in $[1-2\varepsilon, 1/(1+\varepsilon)]$. Bei $\alpha = 1-2\varepsilon$ ist

$$u''(1-2\varepsilon) = -\frac{(1-4\varepsilon)(2\varepsilon^2+1)}{4\varepsilon(1-\varepsilon)^2(1+\varepsilon)(1-4\varepsilon^2)} < 0. \tag{3.48}$$

Das bedeutet, dass für $\alpha \geq 1-2\varepsilon$ die zweite Ableitung von u stets negativ ist.

Die zweite Ableitung $\frac{1}{\alpha(1-\alpha)}$ des Zählers v von r_ε ist positiv. Sowohl u als auch v sind negativ. Damit können wir zeigen, dass r_ε für $\alpha \geq 1-2\varepsilon$ fällt. Denn es gilt

$$r'_\varepsilon = \left(\frac{v}{u}\right)' = \frac{v'u - vu'}{u^2} < 0 \tag{3.49}$$

genau dann, wenn $v'u - vu' < 0$. Da diese Bedingung für $\alpha = 1-2\varepsilon$ erfüllt ist, reicht es zu zeigen dass $v'u - vu'$ fallend ist. Mit $u, v, u'' < 0$ und $v'' > 0$ gilt

$$(v'u - vu')' = v''u - vu'' < 0, \tag{3.50}$$

und damit ist r_ε fallend. Das heißt, es gilt für diesen Fall ebenfalls die Schranke (3.37).

Somit erhalten wir

$$\mathbf{Prob}(F) = O(m^{1-d}) + m \sum_{j=2}^{m/(1+\varepsilon)} O(m^{2-2d}) = O(m^{1-d}). \tag{3.51}$$

□

3.3.2 Analyse der Einfügezeit

Wir wollen hier die erwartete Laufzeit untersuchen, die benötigt wird, um einen neuen Schlüssel x mit Algorithmus CUCKOO-CACHE-BLOCK-INSERT (Algorithmus 3.2 auf Seite 60) in ein Feld T einzufügen. Zu dem Zeitpunkt, da x eingefügt werden soll, befindet sich bereits eine Menge S von n Schlüsseln in T.

Wenn ein neuer Schlüssel x eingefügt wird, gibt es folgende zwei Fälle:

- Einer der beiden Blöcke $y_1 = h_1(x)$ oder $y_2 = h_2(x)$ ist frei. Dann ist der Algorithmus sofort fertig.
- Keiner der beiden Blöcke y_1 und y_2 ist frei. Dann betrachten wir für jede Zelle z_i, $1 \leq i \leq 2d$ in den Blöcken y_1 und y_2 ihren Nachbarn $z_i' = \zeta_B(z_i)$.

Wir werden uns überlegen, mit welcher Wahrscheinlichkeit in dem in Abschnitt 3.2.2 auf Seite 61 eingeführten Graphen eine beliebig gewählte Zelle y eine Entfernung von höchstens k zu einer freien Zelle beziehungsweise zu einem freien Block hat. Wir werden sehen, dass die meisten Zellen eine kurze Entfernung zu einem freien Block haben und es nur sehr wenige Zellen gibt, die eine große Entfernung zu einem freien Block besitzen.

Dazu betrachten wir den Einfügealgorithmus „rückwärts“: Die Menge der freien Blöcke sei Y_0. Ferner sei $X_0 := \{x \in S \mid x \text{ ist in } Y_0 \text{ gespeichert}\}$. Induktiv definieren wir für $k \geq 0$:

$$X_{k+1} := \{x \in S \mid \Gamma_B(x) \cap Y_k \neq \emptyset\} \tag{3.52}$$

als die Menge der Schlüssel, die wenigstens einen Nachbarn in Y_k haben und

$$Y_{k+1} := \{y \in [m] \mid \exists x \in X_{k+1} \colon x \text{ ist in } y \text{ in gespeichert}\} \tag{3.53}$$

als die Menge der Blöcke, die einen Abstand von höchstens $k+1$ zu einem freien Block haben.

Im Folgenden zeigen wir, dass die Mengen $Y_0, Y_1, \ldots$ sehr gut expandieren:

- Wir zeigen zuerst, dass die Mengen Y_i für $i = 0, \ldots, k^*$ jeweils um einen konstanten Faktor größer werden für $|Y_k^*| \leq m\left(\frac{\varepsilon}{1+\varepsilon} + \frac{1}{2}\right)$.
- Anschließend zeigen wir, dass die Größe der Mengen $[m] - Y_{k^*+1}, [m] - Y_{k^*+2}, \ldots$ für $|Y_k^*| \geq m/2$ jeweils um einen konstanten Faktor schrumpft. Dabei ist dieser Faktor $\sqrt{2}$, solange $[m] - |Y_i| \geq \gamma dm$, $\gamma = 2/(e^4 d^3)$, und für $[m] - |Y_i| < \gamma dm$ sogar $d^{2/3}$.

Mit den folgenden Hilfssätzen wollen wir die beiden Expansionseigenschaften der Folge $Y_0, Y_1, \ldots$ zeigen.

Lemma 8. *Sei $\varepsilon \leq 0.1$ und $d \geq 90.1 \cdot \ln \frac{1}{\varepsilon}$. Dann gilt: Mit Wahrscheinlichkeit $1 - O(\beta^m)$, $0 < \beta < 1$, gibt es für jede Blockmenge Y mit $\frac{\varepsilon}{1+\varepsilon} m \leq r = |Y| \leq \frac{5}{13} m$ wenigstens $\frac{4}{3} rd$ Schlüssel, die Y mit h_1 oder h_2 treffen.*

Beweis. Eine Blockmenge Y, $|Y| = r$, wird genau dann von $\frac{4}{3}rd$ Schlüsseln mit wenigstens einer Hashfunktion getroffen, wenn höchstens $n - \frac{4}{3}rd$ Schlüssel mit beiden Hashfunktionen Y *nicht* treffen.

Es genügt zu zeigen, dass die Wahrscheinlichkeit für die Existenz einer solchen Menge Y der Größe $r \leq \frac{5}{13}m$ höchstens $O(\beta'^m)$ für eine Konstante $\beta' < 1$ ist.

Die Wahrscheinlichkeit dafür, dass mehr als $n - \frac{4}{3}rd$ Schlüssel Y mit beiden Hashfunktionen nicht treffen, schätzen wir wieder mit Hilfe der Chernoff-Hoeffding-Schranke ab. Sei dazu

$$I_x := \begin{cases} 1, & \text{falls } h_1(x), h_2(x) \notin Y, \\ 0, & \text{sonst} \end{cases} \tag{3.54}$$

und $I := \sum_{x \in S} I_x$. Wir sehen:

$$\mathbf{E}(I) = n\left(1 - \frac{r}{m}\right)^2 \tag{3.55}$$

und

$$\mathbf{Prob}(I \geq n - \frac{4}{3}rd) \leq \left(\frac{n\left(1-\frac{r}{m}\right)^2}{n - \frac{4}{3}rd}\right)^{n-\frac{4}{3}rd} \left(\frac{n - n\left(1-\frac{r}{m}\right)^2}{\frac{4}{3}rd}\right)^{\frac{4}{3}rd} \tag{3.56}$$

Da es $\binom{m}{r}$ Möglichkeiten gibt, ein Y der Größe r zu wählen, erhalten wir als obere Wahrscheinlichkeitsschranke für das Ereignis $F(r)$, dass eine Blockmenge $\overline{Y}$ der Größe $m - r$ existiert, in die $n - \frac{4}{3}rd$ Schlüssel mit beiden Hashfunktionen treffen,

$$\mathbf{Prob}(F(r)) \leq \binom{m}{r} \left(\frac{n\left(1-\frac{r}{m}\right)^2}{n - \frac{4}{3}rd}\right)^{n-\frac{4}{3}rd} \left(\frac{n - n\left(1-\frac{r}{m}\right)^2}{\frac{4}{3}rd}\right)^{\frac{4}{3}rd}. \tag{3.57}$$

Mit $n = dm/(1+\varepsilon)$ finden wir

$$n - n\left(1 - \frac{r}{m}\right)^2 = \frac{rd}{1+\varepsilon}\left(2 - \frac{r}{m}\right) \tag{3.58}$$

und

$$\frac{n\left(1-\frac{r}{m}\right)^2}{n - \frac{4}{3}rd} = \frac{\left(1-\frac{r}{m}\right)^2}{1 - \frac{4}{3}(1+\varepsilon)\frac{r}{m}}. \tag{3.59}$$

Das führt uns schließlich mit Ungleichung (1.5) zu

$$\mathbf{Prob}(F(r)) \leq \frac{m^m}{r^r(m-r)^{m-r}} \left(\frac{\left(1-\frac{r}{m}\right)^2}{1 - \frac{4}{3}(1+\varepsilon)\frac{r}{m}}\right)^{\frac{md}{1+\varepsilon} - \frac{4}{3}rd} \left(\frac{2 - \frac{r}{m}}{\frac{4}{3}(1+\varepsilon)}\right)^{\frac{4}{3}rd}. \tag{3.60}$$

Setzen wir $r = \alpha m$, erhalten wir für diesen Ausdruck eine obere Schranke:

$$\mathbf{Prob}(F(r)) \leq \left[\frac{1}{\alpha^\alpha (1-\alpha)^{1-\alpha}} \left(\frac{(1-\alpha)^2}{1-(1+\varepsilon)\frac{4}{3}\alpha} \right)^{\frac{d}{1+\varepsilon}-\frac{4}{3}\alpha d} \left(\frac{2-\alpha}{\frac{4}{3}(1+\varepsilon)} \right)^{\frac{4}{3}\alpha d} \right]^m. \tag{3.61}$$

Es reicht zu zeigen, dass der Ausdruck

$$\frac{1}{\alpha^\alpha (1-\alpha)^{1-\alpha}} \left(\frac{(1-\alpha)^2}{1-(1+\varepsilon)\frac{4}{3}\alpha} \right)^{\frac{d}{1+\varepsilon}-\frac{4}{3}\alpha d} \left(\frac{2-\alpha}{\frac{4}{3}(1+\varepsilon)} \right)^{\frac{4}{3}\alpha d} \tag{3.62}$$

für $\varepsilon/(1+\varepsilon) \leq \alpha \leq 5/13$ kleiner als 1 ist. Das ist genau dann der Fall, wenn d größer ist als

$$\frac{\alpha \ln \alpha + (1-\alpha)\ln(1-\alpha)}{\left(\frac{1}{1+\varepsilon} - \frac{4}{3}\alpha\right)\left(2\ln(1-\alpha) - \ln\left(1 - \frac{4(1+\varepsilon)\alpha}{3}\right)\right) + \frac{4\alpha}{3}\left(\ln(2-\alpha) - \ln\left(\frac{4(1+\varepsilon)}{3}\right)\right)}. \tag{3.63}$$

Um ein passendes d zu ermitteln, untersuchen wir den Bruch (3.63) mit dem Zähler u und dem Nenner v genauer. Wir zeigen, dass v eine positive zweite Ableitung für $0 < \alpha \leq \frac{5}{13}$ hat, wenn $\varepsilon \leq 0.1$ ist. Wenn wir also in diesem Bruch den Nenner durch eine Sekante $\tilde{v}$ ersetzen, welche v bei $\alpha = 0$ und $\alpha = \frac{5}{13}$ schneidet, ist $u/\tilde{v}$ eine obere Schranke für u/v (siehe Abbildung 3.8 auf der nächsten Seite).

Die zweite Ableitung des Nenners v ist

$$v'' = \frac{12\alpha - 16\varepsilon - 8 + 84\alpha\varepsilon + 10\alpha^2 - 40\alpha^2\varepsilon - 32\alpha^2\varepsilon^2 + 64\varepsilon^2 - 12\alpha^3\varepsilon - 12\alpha^3}{3(2-\alpha)^2(1+\varepsilon)(-3+4\alpha+4\alpha\varepsilon)(1-\alpha)^2}. \tag{3.64}$$

Der Nenner dieses Terms ist für $\alpha < \frac{5}{13}$ negativ. Um nun eine Aussage über das Verhalten von v'' zu bekommen, reicht es, die Nullstellen des Zählers zu betrachten. Da der Zähler ein Polynom dritten Grades in α ist, kann man die Nullstellen exakt ermitteln. Das ist uns aber zu aufwändig, stattdessen betrachten wir die Ableitung des Zählers. Diese ist

$$12 + 84\varepsilon + 20\alpha - 80\alpha\varepsilon - 64\varepsilon^2\alpha - 36\alpha^2\varepsilon - 36\alpha^2, \tag{3.65}$$

und die beiden Nullstellen dieses Terms sind

$$\alpha_{1,2} = \frac{5 - 20\varepsilon - 16\varepsilon^2 \pm \sqrt{133 + 664\varepsilon + 996\varepsilon^2 + 640\varepsilon^3 + 256\varepsilon^4}}{18(1+\varepsilon)}. \tag{3.66}$$

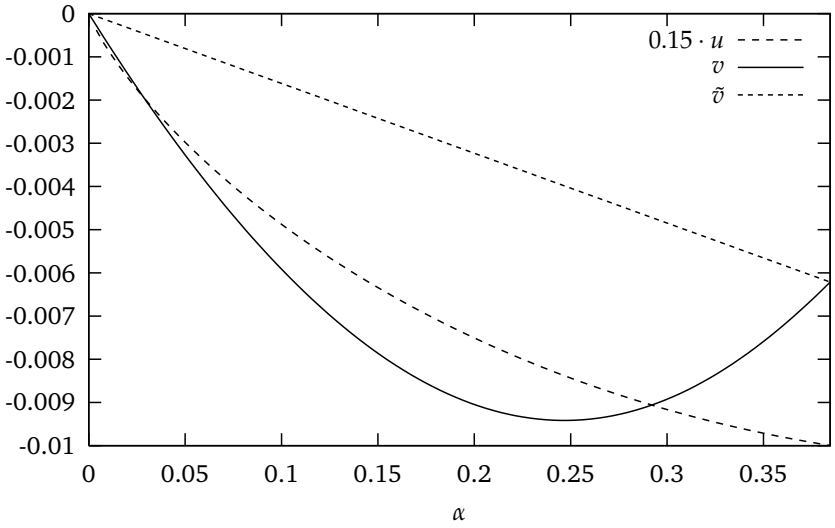


Abbildung 3.8: Illustration von Zähler u und Nenner v des Bruches von (3.63) und Sekante $\tilde{v}$ von v, $\varepsilon = 0.01$

Man überzeugt sich leicht davon, dass eine Nullstelle stets negativ ist und die andere liegt für $0 < \varepsilon \leq 0.1$ jenseits von $\frac{5}{13}$. Setzen wir $\alpha = \frac{5}{13}$ in den Ausdruck (3.65) ein, erhalten wir

$$\frac{2428}{169} + \frac{8096}{169}\varepsilon - \frac{320}{13}\varepsilon^2. \tag{3.67}$$

Dieser Ausdruck ist für $\varepsilon < 0.1$ positiv. Weil also die Ableitung des Zählers von v'' bei $\alpha = \frac{5}{13}$ positiv ist und für $0 \leq \alpha \leq \frac{5}{13}$ keine Nullstelle hat, ist der Zähler in diesem Bereich überall steigend. Setzen wir $\alpha = \frac{5}{13}$ in den Zähler von v'' ein, erhalten wir

$$-\frac{5686}{2197} + \frac{21328}{2197}\varepsilon + \frac{10016}{169}\varepsilon^2. \tag{3.68}$$

Dieser Ausdruck ist für $\varepsilon \leq 0.1$ negativ. Weil der Zähler, wie gezeigt, für $0 \leq \alpha \leq \frac{5}{13}$ steigend ist, ist er in diesem Bereich negativ. Da der Nenner von v'' ebenfalls negativ ist, ist v'' also positiv.

Die Sekante $\tilde{v}$, die v bei $\alpha = 0$ und $\alpha = \frac{5}{13}$ schneidet, hat die Geradengleichung

$$\tilde{v}(\alpha) = \frac{13}{5} v\left(\frac{5}{13}\right) \cdot \alpha \tag{3.69}$$

Ersetzen wir schließlich v durch $\tilde{v}$ im Bruch (3.63), erhalten wir

$$\frac{5(\alpha \ln(\alpha) + (1-\alpha)\ln(1-\alpha))}{13\alpha\left(\left(\frac{1}{1+\varepsilon} - \frac{20}{39}\right)\ln\frac{192}{13(19-20\varepsilon)} + \frac{20}{39}\ln\frac{63}{52(1+\varepsilon)}\right)}. \tag{3.70}$$

Der Ausdruck

$$\left(\left(\frac{1}{1+\varepsilon}-\frac{20}{39}\right)\ln\frac{192}{13(19-20\varepsilon)}+\frac{20}{39}\ln\frac{63}{52(1+\varepsilon)}\right) \tag{3.71}$$

für $\varepsilon \leq 0.1$ negativ. Der Ausdruck

$$\frac{\alpha\ln(\alpha)+(1-\alpha)\ln(1-\alpha)}{\alpha} \tag{3.72}$$

ist ebenfalls negativ und hat eine positive Ableitung. Damit ist der Ausdruck (3.70) fallend und nimmt sein Maximum an der linken Grenze des Bereiches für α an. Da $\alpha \geq \frac{\varepsilon}{1+\varepsilon}$, ist dieser Ausdruck höchstens

$$\frac{5(\varepsilon\ln(\varepsilon)-(1+\varepsilon)\ln(1+\varepsilon))}{13\varepsilon\left(\left(\frac{1}{1+\varepsilon}-\frac{20}{39}\right)\ln\frac{192}{13(19-20\varepsilon)}+\frac{20}{39}\ln\frac{63}{52(1+\varepsilon)}\right)} =: g_\varepsilon \tag{3.73}$$

Wählen wir also d so groß wie dieser Ausdruck ist, erhalten wir $\mathbf{Prob}(F(r)) = O(\beta'^m)$ für eine Zahl $\beta' < 1$.

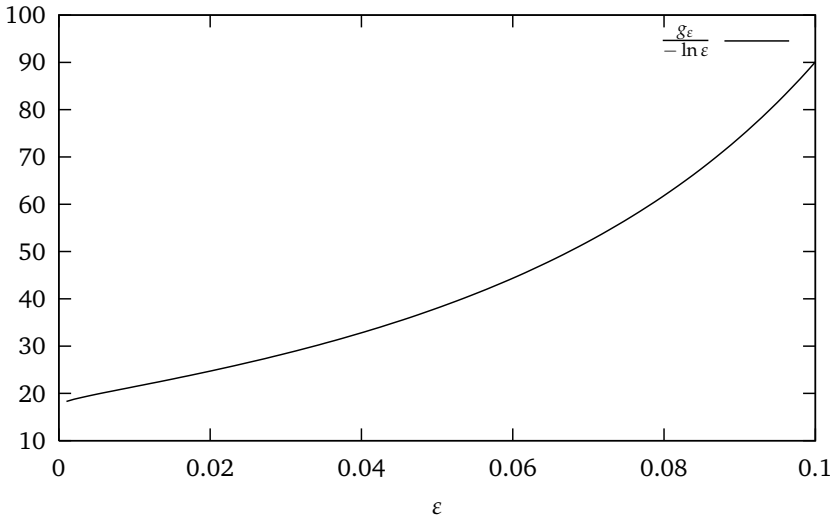


Abbildung 3.9: Verlauf der Funktion $\frac{g_\varepsilon}{-\ln\varepsilon}$

Wir betrachten Ausdruck g_ε noch ein wenig genauer. Zunächst sehen wir, dass

$$\lim_{\varepsilon\to 0}\frac{g_\varepsilon}{-\ln\varepsilon} = -\frac{15}{74\ln 2-39\ln 13-19\ln 19+59\ln 3+20\ln 7} = 15.82\ldots \tag{3.74}$$

Auf die weitergehende Analyse von g_ε wollen wir hier verzichten. Statt dessen sei auf Abbildung 3.9 auf der vorherigen Seite verwiesen. Offensichtlich ist $\frac{g_\varepsilon}{-\ln \varepsilon}$ eine stetige Funktion. In Abbildung 3.9 sehen wir auch, dass diese Funktion wächst. Für $\varepsilon = 0.1$ erhalten wir den Wert $90.08177\ldots$. □

Lemma 9. *Es gelte die in Lemma 8 behauptete Eigenschaft: Für jede Blockmenge Y mit $\frac{\varepsilon}{1+\varepsilon}m \leq r = |Y| \leq \frac{5}{13}m$ gibt es wenigstens $\frac{4}{3}rd$ Schlüssel, die Y mit h_1 oder h_2 treffen. Dann gibt es ein $k^* < 1 + \log_{4/3} \frac{1+\varepsilon}{2\varepsilon}$, so dass $|Y_{k^*}| > \frac{m}{2}$.*

Beweis. Betrachten wir eine Blockmenge Y_k mit $|Y_k| \leq \frac{5}{13}m$ sowie die Menge X_{k+1} der Schlüssel, die einen Nachbarn in Y_k haben. Jeder Schlüssel $x \in X_{k+1}$ ist in Y_{k+1} gespeichert, denn falls x nicht in einem Block $y \in Y_k$ gespeichert ist, dann kann x mit der alternativen Hashfunktion aus y heraus in einen Block, der in Y_k liegt, verschoben werden. Das heißt aber, dass $x \in Y_{k+1}$.

Da in jeder Menge $Y_0, Y_1, \ldots$ genau εn freie Zellen liegen, ist $|Y_0| \geq \frac{\varepsilon n}{d} = \frac{\varepsilon}{1+\varepsilon}m$. Wenn die Behauptung von Lemma 8 auf Seite 76 wahr ist, folgt, dass $X_{k+1} \geq \frac{4d}{3}|Y_k|$, falls $|Y_k| \leq \frac{5}{13}m$, und somit

$$|Y_{k+1}| \geq \frac{\frac{4d}{3}|Y_k| + \varepsilon n}{d}| \geq \frac{4}{3}|Y_k| + \frac{\varepsilon m}{1+\varepsilon} > \frac{4}{3}|Y_k|. \tag{3.75}$$

und

$$|Y_k| \geq \left(\frac{4}{3}\right)^k \frac{\varepsilon m}{1+\varepsilon}. \tag{3.76}$$

Sei k' das kleinste k, für das $|Y_k| > \frac{5}{13}m$ gilt. Dann ist entweder $|Y_{k'}| > m/2$ oder $|Y_{k'+1}| \geq \frac{4}{3} \cdot \frac{5m}{13} > m/2$. Im ersten Fall wählen wir $k^* = k'$ und im zweiten Fall wählen wir $k^* = k' + 1$. Aus (3.76) folgt, dass $k' \leq \left\lceil \log_{4/3} \frac{5(1+\varepsilon)}{13\varepsilon} \right\rceil$ und somit $k^* \leq 2 + \log_{4/3} \frac{5(1+\varepsilon)}{13\varepsilon} < 1 + \log_{4/3} \frac{1+\varepsilon}{2\varepsilon}$. □

Lemma 10. *Sei $d \geq 40$. Mit Wahrscheinlichkeit $1 - O(\beta^m)$, $\beta < 1$, gibt es für jedes Y mit $\frac{m}{2} < |Y| \leq m - \frac{2m}{e^4 d^3}$ wenigstens $n - \frac{2d}{3}(m - |Y|)$ Schlüssel, die Y mit h_1 oder h_2 treffen.*

Beweis. Die Menge Y wird genau dann von mindestens $n - \frac{2d}{3}(m - |Y|)$ Schlüsseln getroffen, wenn höchstens $\frac{2d}{3}(m - |Y|) = \frac{2d}{3}|\overline{Y}|$ Schlüssel mit *beiden* Hashfunktionen in die Menge $\overline{Y}$ fallen. Es genügt zu zeigen, dass die Wahrscheinlichkeit für die Existenz einer solchen Menge $\overline{Y}$ der Größe $\frac{2m}{e^4 d^3} \leq r < m/2$ höchstens $O(\beta'^m)$ für eine Konstante $\beta' < 1$ ist.

Die Wahrscheinlichkeit für die Existenz einer solchen Menge $\overline{Y}$ der Größe r mit $\frac{2m}{e^4d^3} \leq r \leq \frac{m}{2}$ schätzen wir wieder mit der Chernoff-Hoeffding-Schranke (1.9) ab. Wir definieren

$$I_x := \begin{cases} 1, & \text{falls } h_1(x), h_2(x) \in \overline{Y}, \\ 0 & \text{sonst.} \end{cases} \tag{3.77}$$

Mit $I := \sum_{x \in S} I_x$ und $\mathbf{Prob}(I_x = 1) = (r/m)^2$ erhalten wir unter Verwendung der Beziehung $n = \frac{d}{1+\varepsilon}m$:

$$\mathbf{E}(I) = n\left(\frac{r}{m}\right)^2 = \frac{dr^2}{(1+\varepsilon)m} \tag{3.78}$$

und schließlich

$$\begin{aligned} \mathbf{Prob}(I \geq \frac{2}{3}rd) &\leq \left(\frac{dr^2}{(1+\varepsilon)m\frac{2}{3}rd}\right)^{\frac{2}{3}rd} \left(\frac{\frac{dm}{1+\varepsilon} - \frac{dr^2}{(1+\varepsilon)m}}{\frac{dm}{1+\varepsilon} - \frac{2}{3}rd}\right)^{\frac{dm}{1+\varepsilon}-\frac{2}{3}rd} \\ &= \left(\frac{3r}{2(1+\varepsilon)m}\right)^{\frac{2}{3}rd} \left(\frac{3(m^2-r^2)}{(3m-2r(1+\varepsilon))m}\right)^{\frac{dm}{1+\varepsilon}-\frac{2}{3}rd} \end{aligned} \tag{3.79}$$

Für ein festes r gibt es $\binom{m}{r}$ Möglichkeiten für $\overline{Y}$, und wir erhalten für die Wahrscheinlichkeit dafür, dass es eine Menge $\overline{Y}$ mit $r = |\overline{Y}|$ gibt, die von wenigstens $\frac{3}{2}rd$ Schlüsseln mit beiden Hashfunktionen getroffen wird, als obere Schranke

$$\binom{m}{r}\left(\frac{3r}{2(1+\varepsilon)m}\right)^{\frac{2}{3}rd} \left(\frac{3(m^2-r^2)}{(3m-2r(1+\varepsilon))m}\right)^{\frac{dm}{1+\varepsilon}-\frac{2}{3}rd} \tag{3.80}$$

Mit Ungleichung (1.5) können wir für diesen Ausdruck eine obere Schranke $F_\varepsilon(r)$ angeben, nämlich

$$\frac{m^m}{r^r(m-r)^{m-r}}\left(\frac{3r}{2(1+\varepsilon)m}\right)^{\frac{2}{3}rd} \left(\frac{3(m^2-r^2)}{(3m-2r(1+\varepsilon))m}\right)^{\frac{dm}{1+\varepsilon}-\frac{2}{3}rd} =: F_\varepsilon(r) \tag{3.81}$$

Zuerst zeigen wir, dass $F_\varepsilon(r)$ in ε fällt. Dazu betrachten wir

$$\frac{\partial \ln F_\varepsilon(r)}{\partial \varepsilon} = -\frac{dm}{(1+\varepsilon)^2} \ln \frac{3(m^2-r^2)}{3m^2 - 2(1+\varepsilon)rm} \tag{3.82}$$

Dieser Ausdruck ist für $r \leq m/2$ negativ, denn aus $\frac{r}{m} \leq \frac{1}{2} < \frac{2(1+\varepsilon)}{3}$ folgt

$$3r^2 < 2(1+\varepsilon)rm \quad \Rightarrow \quad 3(m^2-r^2) > 3m^2 - 2(1+\varepsilon)rm. \tag{3.83}$$

Damit ist

$$F_\varepsilon(r) < \frac{m^m}{r^r(m-r)^{m-r}} \left(\frac{3r}{2m}\right)^{\frac{2}{3}rd} \left(\frac{3(m^2-r^2)}{(3m-2r)m}\right)^{dm-\frac{2}{3}rd} =: g_1(r). \quad (3.84)$$

Nun zeigen wir, dass der Ausdruck $g_1(r)$ in d fällt. Dazu setzen wir $r = \alpha m$ und bilden die erste Ableitung von $\ln g_1(r)$ nach d:

$$\frac{1}{m} \cdot \frac{\partial \ln g_1(r)}{\partial d} = \ln 3 + \frac{2}{3}\alpha \ln \frac{\alpha}{2} + \left(1 - \frac{2}{3}\alpha\right) \ln \frac{1-\alpha^2}{3-2\alpha} =: g_2(\alpha) \quad (3.85)$$

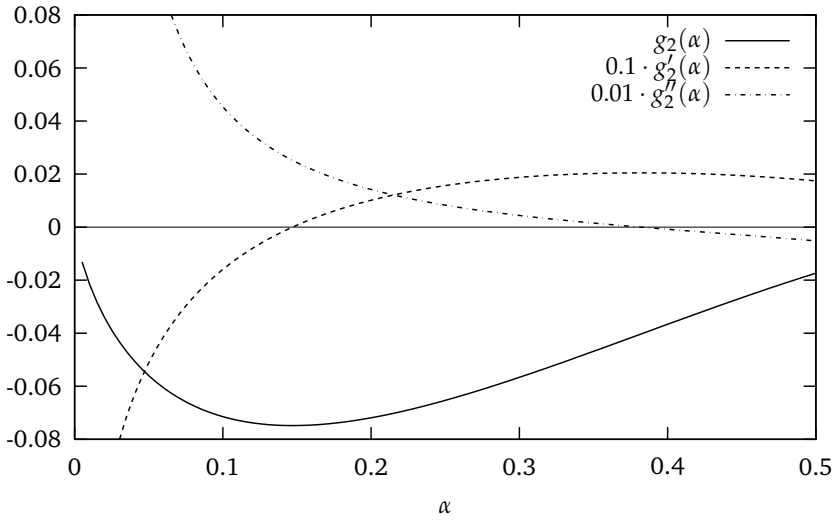


Abbildung 3.10: Verlauf von $g_2(\alpha)$, $g_2'(\alpha)$ und $g_2''(\alpha)$ für $0 < \alpha \leq \frac{1}{2}$

Um zu zeigen, dass g_1 in d fällt, reicht es zu zeigen, dass $g_2(\alpha)$ für $0 < \alpha \leq 1/2$ negativ ist. Dazu berechnen wir

$$g_2'(\alpha) = \frac{2}{3} \cdot \left(2 + \ln \frac{\alpha}{2} - \ln \frac{1-\alpha^2}{3-2\alpha} - \frac{(3-2\alpha)\alpha}{1-\alpha^2}\right) \quad (3.86)$$

und

$$g_2''(\alpha) = \frac{2}{3} \cdot \frac{3 - 13\alpha + 18\alpha^2 - 13\alpha^3 + 3\alpha^4}{\alpha(1-\alpha^2)^2(3-2\alpha)} \quad (3.87)$$

Für $0 \leq \alpha \leq 1/2$ hat g_2'' genau eine Nullstelle, nämlich bei $\alpha = 0.3819\ldots$. Das heißt, dass g_2' an dieser Stelle eine Extremstelle besitzt. Diese ist auch die einzige für $0 < \alpha \leq 1/2$. Da $g_2''(0) > 0$ und $g_2''(1/2) < 0$, ist diese Extremstelle ein Maximum. Weil ferner $g_2'(0) < 0$ und $g_2'(1/2) > 0$ und g_2'

zwischen 0 und $1/2$ genau eine Maximumstelle hat, besitzt g_2' in diesem Intervall genau eine Nullstelle $\hat{\alpha}$. Daraus folgt, dass $\hat{\alpha}$ die einzige Extremstelle von g_2 im Intervall $0 < \alpha \leq 1/2$ ist. Da $g_2'(0) < 0$ und $g_2'(1/2) > 0$, ist $\hat{\alpha}$ eine Minimumstelle von g_2. Weil nun $\lim_{\alpha \to 0} g_2(\alpha) = -0$ und $g_2(1/2) < 0$, folgt $g_2(\alpha) < 0$ für $0 < \alpha \leq 1/2$. Also fällt g_1 in d. Wenn wir die Behauptung für $d = 40$ zeigen, gilt sie für alle $d \geq 40$.

Betrachten wir nun $g_1(r)$. Für $r = \alpha m$ erhalten wir:

$$\frac{\partial \ln g_1}{m \cdot \partial r} = -\ln(\alpha) + \ln(1-\alpha) + \frac{2d}{3}\left(\ln\frac{\alpha}{2} + \ln\frac{3-2\alpha}{1-\alpha^2} - 3\alpha + \frac{2-3\alpha^3}{1-\alpha^2}\right) =: r_1(\alpha) \quad (3.88)$$

und

$$\frac{\partial^2 \ln g_1}{m \cdot \partial r^2} = \frac{2(d-1)\alpha^4 + (1-\frac{26d}{3})\alpha^3 + (5+12d)\alpha^2 - \left(1+\frac{26d}{3}\right)\alpha + 2d - 3}{(2\alpha-3)\alpha(1-\alpha^2)^2} =: r_2(\alpha) \quad (3.89)$$

Können wir zeigen, dass dieser Ausdruck r_2 genau eine Nullstelle α' für $0 < \alpha \leq 1/2$ besitzt, können wir wie eben argumentieren: Wir sehen, dass $r_2(0) > 0$ und $r_2(1/2) < 0$. Damit ist α' die einzige Maximumstelle für r_1. Da $r_1(0) < 0$ und $r_1(1/2) > 0$, hat $\ln g_1(\alpha m)$ genau eine Minimumstelle α'' für $0 < \alpha \leq 1/2$. Das heißt, dass g_1 sein Maximum an einer der beiden Intervallgrenzen annimmt. Für $\frac{2m}{e^4 d^3} \leq m/2$ ist somit

$$g_1(r) \leq \max\left\{g_1\left(\frac{2m}{e^4 d^3}\right), g_1(m/2)\right\}. \quad (3.90)$$

Nun ist die Funktion $\tilde{g}(\alpha) := \frac{1}{m}\ln g_1(\alpha m)$ unabhängig von m und es gilt

$$\lim_{\alpha \to 0}\left(\frac{1}{m}\ln g_1(\alpha m)\right) = -0. \quad (3.91)$$

Da außerdem $\tilde{g}'(0) = r_1(0) < 0$, ist $\tilde{g}\left(\frac{4}{e^4 d^3}\right) < -c$ für eine Konstante $c > 0$ und damit

$$g_1\left(\frac{2m}{e^4 d^3}\right) = O\left(e^{-c}\right). \quad (3.92)$$

Für $d = 40$ ist

$$g_1(m/2) = (0.99826867\ldots)^m, \quad (3.93)$$

woraus schließlich die Behauptung von Lemma 10 auf Seite 81 folgt.

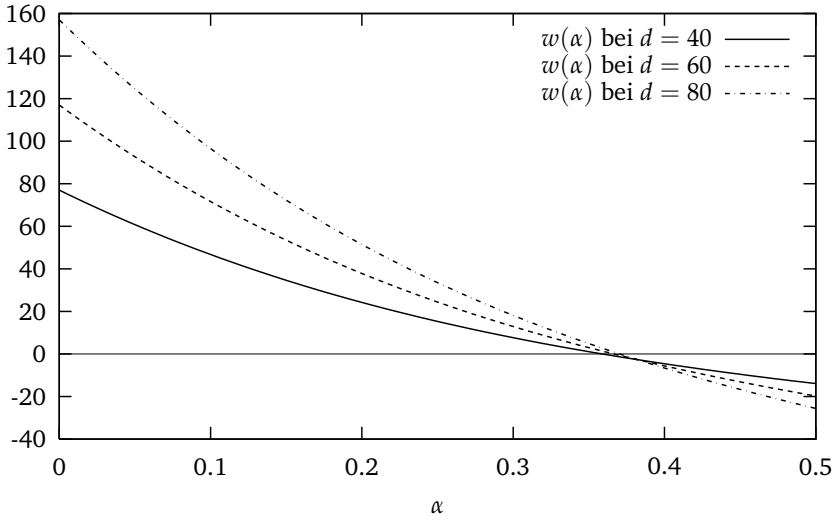


Abbildung 3.11: Verlauf von $w(\alpha)$ für verschiedene d

Bleibt noch zu zeigen, dass r_2 genau eine Nullstelle α' für $0 < \alpha \leq 1/2$ besitzt. Um zu vermeiden, ein Polynom vierten Grades zu lösen, betrachten wir den Zähler $w(\alpha) = 2(d-1)\alpha^4 + (1 - \frac{26d}{3})\alpha^3 + (5+12d)\alpha^2 - \left(1 + \frac{26d}{3}\right)\alpha + 2d - 3$ des Ausdruckes (3.89) und berechnen

$$w' = 8(d-1)\alpha^3 + (3-26d)\alpha^2 + (10+24d)\alpha - 1 - \frac{26d}{3} \tag{3.94}$$

und

$$w'' = 24(d-1)\alpha^2 + (6-52d)\alpha + 10 + 24d \tag{3.95}$$

Die Nullstellen für w'' sind

$$\begin{aligned} w_1 &= \frac{26d - 3 - \sqrt{100d^2 + 180d + 249}}{24(d-1)} \\ w_2 &= \frac{26d - 3 + \sqrt{100d^2 + 180d + 249}}{24(d-1)} \end{aligned} \tag{3.96}$$

Während für $d \geq 40$ offensichtlich $w_2 > 1$ gilt, müssen wir w_1 genauer betrachten. Aus $0 < 96d^2 + 72d + 88$ folgt $100d^2 + 180d + 169 < (14d+9)^2$ und weiter $\sqrt{180d + 100d^2 + 169} < 26d + 3 - 12(d-1)$ und schließlich $w_1 = \frac{26d+3-\sqrt{180d+100d^2+169}}{24(d-1)} > 1/2$. Also hat w'' keine Nullstelle in $0 < \alpha \leq 1/2$. Da $w''(0) > 0$, ist $w''(\alpha) > 0$. Das heißt, dass $w'(\alpha)$ für $0 < \alpha \leq 1/2$ steigend ist. Da $w'(1/2) < 0$, ist auch $w'(\alpha) < 0$ für $0 < \alpha \leq 1/2$, das

heißt, w ist monoton. Weil $w(0) > 0$ und $w(1/2) < 0$, hat w genau eine Nullstelle $\alpha' \in (0, 1/2]$. □

Lemma 11. *Es gelte die in Lemma 10 behauptete Eigenschaft: Für jedes Y mit $\frac{m}{2} < |Y| \leq m - \frac{2m}{e^4 d^3}$ gibt es wenigstens $n - \frac{2d}{3}(m - |Y|)$ Schlüssel, die Y mit h_1 oder h_2 treffen. Es gelte ferner $|Y_{k^*}| > m/2$. Dann gibt es ein $\ell^* \leq 1 + \log_{3/2} \frac{e^4 d^3}{4} \in O(\log d)$, so dass $|m - Y_{k^*+\ell^*}| \leq \frac{2m}{e^4 d^3}$.*

Beweis. Der Beweis funktioniert analog zum Beweis von Lemma 9. Wenn $\frac{2m}{e^4 d^3} < |m - Y_{k^*+\ell}| < m/2$, dann gilt nach Lemma 10

$$|X_{k^*+\ell+1}| \geq n - \frac{2d}{3}(m - |Y_{k^*+\ell}|). \tag{3.97}$$

Alle Schlüssel aus $X_{k^*+\ell+1}$ haben einen Nachbarn in $Y_{k^*+\ell+1}$. Da $Y_{k^*+\ell+1}$ außerdem εn freie Zellen enthält, ist

$$|Y_{k^*+\ell+1}| \geq \frac{\varepsilon n + |X_{k^*+\ell+1}|}{d} \geq m - \frac{2}{3}(m - |Y_{k^*+\ell}|) \tag{3.98}$$

und damit

$$m - |Y_{k^*+\ell+1}| \leq \frac{2}{3}(m - |Y_{k^*+\ell}|) \tag{3.99}$$

und

$$m - |Y_{k^*+\ell}| \leq \left(\frac{2}{3}\right)^{\ell} (m - |Y_{k^*}|) < \frac{m}{2}\left(\frac{2}{3}\right)^{\ell} \tag{3.100}$$

Sei ℓ^* das kleinste ℓ, für das $m - |Y_{k^*+\ell}| \leq \frac{2m}{e^4 d^3}$ gilt. Aus (3.100) folgt, dass $\ell^* \leq \left\lceil \log_{3/2} \frac{e^4 d^3}{4} \right\rceil$. □

Schließlich zeigen wir eine letzte für uns wichtige Eigenschaft von bipartiten Zufallsgraphen (siehe [MR95, Seite 109]):

Lemma 12. *Sei $d \geq 8$ und $\gamma = \frac{2}{e^4 d^3}$. Mit Wahrscheinlichkeit $1 - O(m^{-d/2})$ trifft jede Schlüsselmenge $X \subseteq S$ mit $d \leq |X| \leq \gamma dm$ wenigstens $d^{-1/3}|X|$ verschiedene Blöcke, das heißt $\Gamma_B(X) \geq d^{-1/3}|X|$.*

Beweis. Zur Abkürzung schreiben wir $\Delta := d^{-1/3}$. Für $d \leq j \leq \gamma dm$ sei $F(j)$ das Ereignis, dass es eine Menge X von j Schlüsseln, $d \leq j \leq \gamma dm$, die höchstens Δj Blöcke trifft. Dann gilt

$$\mathbf{Prob}(F(j)) \leq \binom{n}{j}\binom{m}{\lfloor \Delta j \rfloor}\left(\frac{\lfloor \Delta j \rfloor}{m}\right)^{2j} \leq \left(\frac{en}{j}\right)^{j}\left(\frac{em}{\Delta j}\right)^{\Delta j}\left(\frac{\Delta j}{m}\right)^{2j}. \tag{3.101}$$

Die letzte Ungleichung folgt aus (1.6). Wir erhalten für die rechte Seite der letzten Ungleichung

$$\left(\frac{en}{j}\left(\frac{em}{\Delta j}\right)^{\Delta}\left(\frac{\Delta j}{m}\right)^{2}\right)^{j} < \left(ed\cdot e^{\Delta}\cdot\Delta^{2-\Delta}\left(\frac{j}{m}\right)^{1-\Delta}\right)^{j} \tag{3.102}$$

Weil für $d \geq 8$ gilt, dass $\Delta \leq 1/2$, folgt $e^{1+\Delta} \leq e^{3/2}$ und $\Delta^{2-\Delta} < (1/2)^{3/2}$ und $(j/m)^{1-\Delta} < (j/m)^{1/2}$. Daher ergibt sich schließlich

$$\mathbf{Prob}(F(j)) < \left(\left(\frac{e}{2}\right)^{3/2}\cdot d\cdot\left(\frac{j}{m}\right)^{1/2}\right)^{j} \tag{3.103}$$

und für die Wahrscheinlichkeit, dass eines der Ereignisse $F(j)$, $d \leq j \leq \gamma dm$, eintritt, ergibt sich als obere Schranke:

$$\sum_{j=d}^{\gamma dm}\mathbf{Prob}(F(j)) < \sum_{j=d}^{\gamma dm}\left(\left(\frac{e}{2}\right)^{3/2}\cdot d\cdot\left(\frac{j}{m}\right)^{1/2}\right)^{j}. \tag{3.104}$$

Die Summanden dieser Summe bilden für $d \leq j \leq \gamma dm$ eine geometrische Folge mit einem Faktor kleiner als $1/2$, denn es gilt für $j \leq \gamma dm$:

$$\begin{aligned}\frac{\left(\left(\frac{e}{2}\right)^{3/2}\cdot d\cdot\left(\frac{j}{m}\right)^{1/2}\right)^{j}}{\left(\left(\frac{e}{2}\right)^{3/2}\cdot d\cdot\left(\frac{j-1}{m}\right)^{1/2}\right)^{j-1}} &= \left(\frac{j^{j-1}}{(j-1)^{j-1}}\right)\sqrt{\frac{j}{m}}\left(\frac{e}{2}\right)^{3/2}\cdot d \\ &< \sqrt{e}\sqrt{\gamma d}\left(\frac{e}{2}\right)^{3/2}\cdot d = \sqrt{e}\frac{\sqrt{2}}{de^{2}}\left(\frac{e}{2}\right)^{3/2}\cdot d = \frac{1}{2}.\end{aligned} \tag{3.105}$$

Das heißt, dass

$$\sum_{j=d}^{\gamma dm}\left(\left(\frac{e}{2}\right)^{3/2}\cdot d\cdot\left(\frac{j}{m}\right)^{1/2}\right)^{j} < 2\left(\left(\frac{e}{2}\right)^{3/2}\cdot d\cdot\left(\frac{d}{m}\right)^{1/2}\right)^{d} = O(m^{-d/2}), \tag{3.106}$$

woraus sich die Behauptung ergibt. □

Lemma 13. *Es gelte die in Lemma 12 behauptete Eigenschaft: Jede Schlüsselmenge $X \subseteq S$ mit $d \leq |X| \leq \gamma dm$ trifft wenigstens $d^{-1/3}|X|$ verschiedene Blöcke. Dann gibt es für jede Menge Y von $r \geq \gamma m$ Blöcken wenigstens $n - d^{1/3}(m-r)$ Schlüssel, die Y mit einer oder beiden Hashfunktionen treffen.*

Beweis. Angenommen, es gibt eine Menge Y von $r \geq \gamma m$ Blöcken, die von weniger als $n - d^{1/3}(m-r)$ Schlüsseln getroffen wird. Sei X die Menge der Schlüssel, die Y nicht treffen. Dann folgt, dass $|X| > d^{1/3}(m - |Y|)$. Andererseits gilt $\Gamma_B(X) \subseteq [m] - Y$, da für alle $x \in X$ gilt: $h_1(x), h_2(x) \notin Y$. Aus 12 folgt aber $[m] - Y \geq |\Gamma_B(X)| \geq d^{-1/3}|X|$, was ein Widerspruch zu $|X| > d^{1/3}(m - |Y|)$ ist. □

Lemma 14. *Es gelte die in Lemma 13 behauptete Eigenschaft: Für jede Menge Y von $r \geq \gamma m$ Blöcken gibt es wenigstens $n - d^{1/3}(m-r)$ Schlüssel, die Y mit einer oder beiden Hashfunktionen treffen. Sei $m - |Y_{k^*+\ell^*}| \leq \gamma m$. Dann gilt $m - |Y_{k^*+\ell^*+j}| \leq d^{-2j/3}\gamma m$ für $j \geq 0$.*

Beweis. Für $j = 0$ ist die Aussage von Lemma 14 wahr. Für $j > 1$ gilt

$$|X_{k^*+\ell^*+j+1}| \geq n - d^{1/3}(m - |Y_{k^*+\ell^*+j}|) \tag{3.107}$$

und damit nach Lemma 13 (analog zum Beweis von Lemma 11)

$$\begin{aligned} m - |Y_{k^*+\ell^*+j+1}| &\leq m - \frac{\varepsilon n + |X_{k^*+\ell^*+j+1}|}{d} \\ &\leq m - \frac{\varepsilon n + n - d^{1/3}(m - |Y_{k^*+\ell^*+j}|)}{d} \leq d^{-2/3}|Y_{k^*+\ell^*+j}| \\ &\leq d^{-2(j+1)/3}\gamma m. \end{aligned} \tag{3.108}$$

□

Zu guter Letzt wollen wir überlegen, wie groß der Erwartungswert für die Anzahl der besuchten Blöcke ist, wenn ein neuer Schlüssel x eingefügt werden soll.

Lemma 15. *Sei $\varepsilon \leq 0.1$ und $d \geq 90.1 \ln(1/\varepsilon)$ mit $d \in O(\ln(1/\varepsilon))$. Ist Z_x die Anzahl der Blöcke, die man betrachten muss, bis man einen freien Block gefunden hat, dann gilt $\mathbf{E}(Z_x) = (1/\varepsilon)^{O(\log d)}$.*

Beweis. Für $\varepsilon \leq 0.1$ und $d \geq -90.1 \ln \varepsilon$ expandieren die Mengen Y_j, $j \geq 0$, wie in den Lemmas 9, 11 und 14 beschrieben mit Wahrscheinlichkeit $1 - O(m^{-d/2})$. Wenn die Expansionseigenschaft verletzt ist, muss ein Rehash durchgeführt werden. Die Kosten eines Rehashes sind höchstens $O(m^2)$, das heißt der Beitrag zu $\mathbf{E}(Z_x)$ in den Fällen, in denen die Y_j nicht wie beschrieben expandieren, ist $O(m^{2-d/2})$.

Gehen wir nun davon aus, dass die Aussagen dieser Lemmas gelten. Mit $k_x = \min\{k \mid h_1(x) \in Y_k \text{ oder } h_2(x) \in Y_k\}$ gilt $Z_x \leq 2\sum_{i=0}^{k_x} d^k < 4d^{k_x}$.

Dann folgt

$$\begin{aligned}\mathbf{E}(Z_x) &\leq \sum_{k\geq 0} 4d^k \,\mathbf{Prob}(k_x \geq k) = 4 + 4\sum_{k\geq 1} d^k \,\mathbf{Prob}(k_x \geq k) \\ &= 4 + 4\sum_{k\geq 1} d^{k+1} \,\mathbf{Prob}(h_1(x), h_2(x) \notin Y_{k-1}) \\ &= 4 + 4d\sum_{k\geq 0} d^k \,\mathbf{Prob}(h_1(x), h_2(x) \notin Y_k) \leq 4 + 4d\sum_{k\geq 0} d^k \left(\frac{m-|Y_k|}{m}\right)^2. \end{aligned} \tag{3.109}$$

Wir teilen die letzte Summe auf und erhalten

$$\sum_{k=0}^{k^*+\ell^*} d^k \left(\frac{m-|Y_k|}{m}\right)^2 < \sum_{k=0}^{k^*+\ell^*} d^k < (k^*+\ell^*+1)d^{k^*+\ell^*} \tag{3.110}$$

und

$$\begin{aligned}\sum_{k>k^*+\ell^*} d^k \left(\frac{m-|Y_k|}{m}\right)^2 &= d^{k^*+\ell^*} \cdot \sum_{k>0} \left(\frac{m-|Y_k^*+\ell^*+k|}{m}\right)^2 \\ &\overset{\text{(Lemma 14)}}{<} d^{k^*+\ell^*} \sum_{k>0} \left(\frac{d^{-2k/3}\gamma m}{m}\right)^2 = d^{k^*+\ell^*}\gamma^2 \sum_{k>0} \left(d^{-4/3}\right)^k \\ &< d^{k^*+\ell^*} \frac{\gamma^2}{1-d^{-4/3}} < d^{k^*+\ell^*}. \end{aligned} \tag{3.111}$$

Wegen $k^* = O(\log(1/\varepsilon))$ (Lemma 9) und $d \in O(\log(1/\varepsilon))$, folgt nach Lemma 11: $k^*+\ell^* = O(\log(1/\varepsilon)+\log d) = O(\log(1/\varepsilon))$. Damit erhalten wir $\mathbf{E}(Z_x) \leq (k^*+\ell^*+2)d^{k^*+\ell^*} = O(\log(1/\varepsilon))d^{O(\log(1/\varepsilon))} = (1/\varepsilon)^{O(\log d)}$. □

3.3.3 Bemerkungen zum Platzbedarf beim Einfügen

Bis jetzt haben wir die erwartete Zahl von zu untersuchenden Blöcken ermittelt, um einen neuen Schlüssel x einzufügen. Wir wollen uns hier am Beispiel CUCKOO-CACHE-BLOCK-INSERT überlegen, wie man die Algorithmen 3.2 auf Seite 60 so modifizieren kann, dass für die Suche nach einer freien Zelle sublinearer Platz ausreicht. Zunächst sehen wir, dass man die Felder *parent* und *visited* mit simplem Cuckoo Hashing realisieren kann, so dass bei r besuchten Blöcken der Platzbedarf nur $O(r)$ beträgt, dafür die erwartete Einfügezeit zwar steigt, aber noch immer eine Konstante ist. Wenn aber r groß wird, etwa $r = O(m)$, steigt der Platzbedarf auch auf $O(m)$.

Um den Platzbedarf zu senken, zählen wir die Blöcke, die aus der Warteschlange Q entfernt worden sind. Diese Zahl heiße ℓ. Man sieht leicht, dass man höchstens $d\ell$ Knoten im Feld *visited* markiert hat und Q somit höchstens $d\ell - \ell = (d-1)\ell$ Knoten enthält. Wir wollen den Algorithmus stoppen, wenn bei $\ell = m^{7/8}$ noch kein freier Block gefunden wurde. An dieser Stelle ist die erste Phase des Verfahrens beendet. Bis dahin benöigt der Algorithmus nur $O(m^{7/8})$ Platz.

In einer zweite Phase betrachten wir der Reihe nach die Blöcke y, die sich noch in Q befinden. Wir schreiben dann kurz: $y \in Q$. Für den Moment wollen wir uns vorstellen, dass wir von jedem dieser Blöcke $y \in Q$ eine Breitensuche mit y als Start durchführen, bis entweder ein freier Block gefunden wurde oder höchstens $m^{7/8}$ Blöcke untersucht wurden.[4] Wie wollen dieses Verfahren *mehrfache Breitensuche* nennen. Diese Suche kann für jedes $y \in Q$ auf dem gleichen Platz durchgeführt werden.

Lemma 16. *Wenn die Voraussetzungen für die Lemmas 9, 11 und 14 erfüllt sind und $m > (1/\varepsilon)^{O(\log d)}$ gilt, untersucht die mehrfache Breitensuche alle Blöcke.*

Beweis. Wir betrachten bei $\Gamma_B(x)$ beginnende Wege im gerichteten Graphen $G = ([m], E)$, der in Abschnitt 3.2.2 auf Seite 61 beschrieben ist. Aus Lemma 14 lässt sich schlussfolgern, dass es ein $L \leq C + \log_{d^{2/3}} m$ mit $C = O(\log(1/\varepsilon))$ gibt, so dass $|Y_L| = m$, das heißt dass alle kürzesten Wege, die zu einem freien Block führen, eine Länge von höchstens L haben.

Sei $\mathrm{dist}(u,v)$ die Länge eines kürzesten Weges von u nach v in G. Da jeder Knoten in G einen Grad von höchstens d hat, gibt es höchstens

$$2\sum_{j=0}^{k} d^j = 2\frac{d^{k+1}-1}{d-1} \overset{(d>3)}{<} d^{k+1} \tag{3.112}$$

Knoten u mit $\min\{\mathrm{dist}(h_1(x),u), \mathrm{dist}(h_2(x),u)\} \leq k$. Für

$$k = k_1 := \left\lfloor \frac{7}{8\ln d} \ln m \right\rfloor - 1 \tag{3.113}$$

ist $d^{k+1} \leq m^{7/8}$, das heißt, dass in der ersten Phase alle Knoten Knoten u mit $\min\{\mathrm{dist}(h_1(x),u), \mathrm{dist}(h_2(x),u)\} \leq k_1$ untersucht werden. Dann gilt aber $\min\{\mathrm{dist}(h_1(x),y), \mathrm{dist}(h_2(x),y)\} > k_1$ für alle $y \in Q$.

[4] Die bisher angelegten Felder *parent* und *visited* werden dabei nicht mehr benutzt. Wir tun so, als würden wir für jedes $y \in Q$ jeweils eine neue Suche starten, ohne bereits erlangtes Wissen zu benutzen.

Ebenso finden wir, dass für jeden Knoten $y \in Q$ in der zweiten Phase alle Knoten y' mit $\text{dist}(y, y') \leq k'$ untersucht werden, wobei

$$\sum_{j=0}^{k'} d^j < d^{k'+1} \tag{3.114}$$

gilt. Wir wählen

$$k' = k_2 := \left\lfloor \frac{7}{8 \ln d} \ln m \right\rfloor - 1 \tag{3.115}$$

und sehen, dass in der zweiten Phase alle Knoten y' mit $\text{dist}(y, y') \leq k_2$ untersucht werden. Da für $y \in Q$ stets $\min\{\text{dist}(h_1(x), y), \text{dist}(h_2(x), y)\} > k_1$ gilt, sind am Ende der zweiten Phasen alle Knoten mit $\min\{\text{dist}(h_1(x), y), \text{dist}(h_2(x), y)\} \leq k_1 + k_2$ untersucht. Nun gilt:

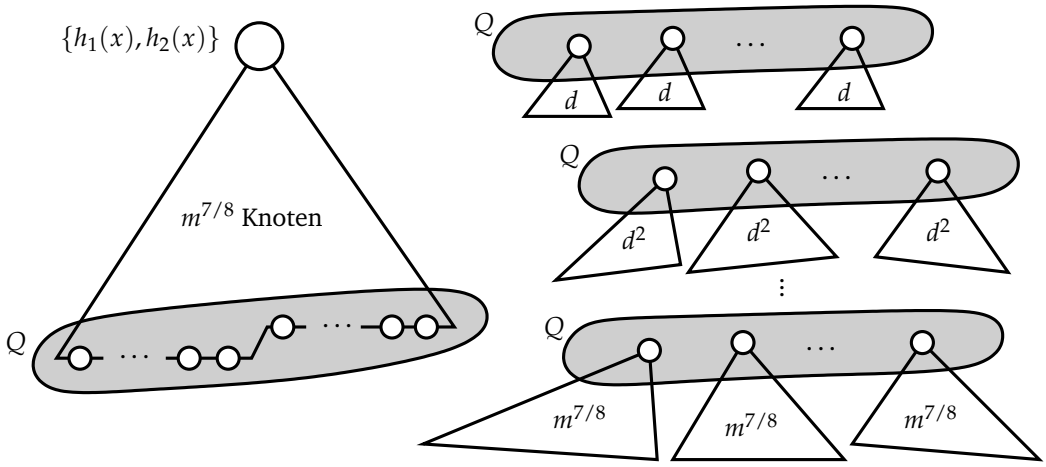
$$\begin{aligned} k_1 + k_2 - L &\geq 2 \left\lfloor \frac{7}{8 \ln d} \ln m \right\rfloor - 2 - (C + \log_{d^{2/3}} m) \\ &\geq \frac{7}{4 \ln d} \ln m - 4 - C - \frac{3}{2 \ln d} \ln m \\ &= \frac{1}{4 \ln d} \ln m - (C + 4) \geq 0, \end{aligned} \tag{3.116}$$

falls $m > d^{4C+4} = d^{O(\log(1/\varepsilon))} = (1/\varepsilon)^{O(\log d)}$. □

Damit die erwartete Laufzeit der mehrfachen Breitensuche konstant ist, darf man nicht nacheinander für jedes $y \in Q$ jeweils $m^{7/8}$ Nachfolgeknoten untersuchen. Stattdessen untersucht man in Runde $j = 1, 2, \ldots$ für jedes $y \in Q$ höchstens d^j (direkte und indirekte) Nachfolger von y (siehe Abbildung 3.12). Dabei untersucht man unter Umständen einige Blöcke mehrfach. Sei wieder Z_x die Anzahl der zu untersuchenden Blöcke um den neuen Schlüssel x einzufügen, wobei Blöcke auch mehrfach gezählt werden dürfen. Wir betrachten zwei Fälle für $k = \min\{j \mid h_1(x) \in Y_k \text{ oder } h_2(x) \in Y_k\}$:

$k \leq k_1$. Es ändert nichts sich gegenüber dem ursprünglichen Algorithmus und es gilt $Z_x \leq 4d^k$ (siehe Beweis von Lemma 15).

$k > k_1$. Zunächst werden höchstens d^{k_1+1} Blöcke untersucht (siehe (3.112)). Nachdem Phase 1 abgeschlossen ist, können höchstens d^{k_1+1} Blöcke aus Q entfernt worden sein, also befinden sich noch höchstens $(d-1) \cdot d^{k_1+1}$ Blöcke in Q. In Phase 2 wird für jedes $j = 1, 2, \ldots$ für jeden der Blöcke $y \in Q$ die Suche gestartet, wobei



Nachdem die Suche nach einer freien Zelle in Phase 1 erfolglos war, befinden sich einige Blöcke in Q. Für jeden dieser Blöcke wird in einer zweiten Phase in Runde $j = 1, 2, \ldots$ jeweils auf dem gleichen Platz ein Weg zu einem freien Block gesucht, wobei höchstens d^j Blöcke betrachtet werden. Dabei reicht es, nur solche j mit $d^j < m^{7/8}$ zu betrachten.

Abbildung 3.12: Platzeffiziente Variante der Einfügeoperation

nach höchstens d^j Blöcken abgebrochen wird. Weil von y startend höchstens alle Blöcke y' mit $\text{dist}(y, y') \leq k' = k - k_1$ untersucht werden müssen, reicht es, $j = 1, \ldots, k' + 1$ zu betrachten (siehe (3.114)). Ein Block, der zum ersten Mal in Runde $j = i$ untersucht wird, wird auch in den Runden $j > i$ untersucht, also höchstens $k' - i + 1$ mal. Damit erhalten wir:

$$\begin{aligned} Z_x &= d^{k_1+1} + (d-1) \cdot d^{k_1+1} \sum_{j=1}^{k'} (k' - j + 1) d^j \\ &= d^{k_1+1} + (d-1) \cdot d^{k_1+1} \left(\frac{d^2}{(d-1)^2} (d^{k'} - 1) - \frac{d}{d-1} k' \right) \\ &= d^{k_1+1} + d^{k_1+2} \left(\frac{d}{d-1} (d^{k'} - 1) - k' \right) \leq d^{k_1+1} + d^{k_1+2} \cdot 2d^{k'} \\ &= d^{k_1+1} + 2d^{k_1+2} d^{k-k_1} \leq 3d^{k+2}. \end{aligned} \quad (3.117)$$

Es bleibt noch, im Beweis von Lemma 15 für Z_x an Stelle von $4d^k$ nun $3d^{k+2}$ zu setzen. Als Ergebnis erhält man wieder einen konstanten Erwartungswert.

3.4 Experimente

Um zu beobachten, wie sich die cachefreundlichen Wörterbücher verglichen mit d-ärem Cuckoo Hashing und Linearem Sondieren verhalten, wurden einige Experimente durchgeführt. Zunächst wurden für die dem Cuckoo Hashing ähnlichen Verfahren die Zahl der Cache-Fehler gemessen. In einer zweiten Messreihe wurde das Laufzeitverhalten der Wörterbuchoperationen untersucht und schließlich wurde getestet, wie viele Schlüssel man bei gegebenem d in einem Wörterbuch speichern kann, und das Resultat wurde mit der in Lemma 7 auf Seite 67 ermittelten Schranke verglichen.

Bevor die Experimente beschrieben werden, wollen wir Implementierungen der Verfahren erläutern.

3.4.1 Anstatt einer Breitensuche

Um den Aufwand für die Breitensuche bei der `insert`-Operation zu vermeiden, wurde stattdessen ein *random walk* implementiert. Um einen neuen Schlüssel x einzufügen, geht man wie folgt vor:

- Beim d-ären Cuckoo Hashing (*cuckoo-d-ary*) wird zufällig ein $i_0 \in [d]$ gewählt und getestet, ob die Zelle $y = h_i(x)$ in Tabelle T_i frei ist. Ist das der Fall, wird x dort gespeichert und man ist fertig. Ansonsten wird x mit $T_i[y]$ vertauscht. Für $j = 1, 2, \ldots$ wird $i_j \in [d] - \{i_{j-1}\}$ gewählt und getestet, ob $T_{i_j}[h_{i_j}(x)]$ frei ist. Ist das der Fall, wird x dort gespeichert und man ist fertig. Ansonsten wird $T_{i_j}[h_{i_j}(x)]$ mit x getauscht. Das ist die gleiche Implementierung, wie sie in [FPSS03] für Experimente benutzt wurde.

- Beim Cuckoo Hashing mit festen Blöcken wird getestet, ob die Blöcke $h_1(x)$ oder $h_2(x)$ frei sind. Wenn das der Fall ist, wird x in einer freien Zelle gespeichert und man ist fertig. Ansonsten wählt man $y_0 \in \{h_1(x), h_2(x)\}$ zufällig. Für $j = 1, 2, \ldots$ wird eine Zelle $z \in y_{j-1}$ zufällig gewählt und $y_j = \zeta_B(z)$ berechnet. Man tauscht x mit $T[z]$ und testet, ob $T[y_j]$ frei ist. Ist das der Fall, wird x in $T[y_j]$ gespeichert und die Einfügeoperation ist abgeschlossen. Um für eine Zelle z eines Blockes y den alternativen Block $\zeta_B(z)$ zu berechnen, bieten sich die folgenden Strategien an:

  - Man berechnet $\zeta_B(z)$ wie in Definition 8 auf Seite 59. Dieses Verfahren heißt *cuckoo-block*.

– man wählt $h_2(x) = q(x) - h_1(x)$ für zwei zufällig gewählte $h_1, q\colon U \to [m]$. Dann gilt nämlich $h_1(x) = q(x) - h_2(x)$ und $\zeta_B(z) = q(T[z]) - y$. Dieses Verfahren heißt *cuckoo-block-improved*.[5]

- Beim Cuckoo Hashing mit begrenzter Sondierungsweite (*cuckoo-lp*) geht man analog zum Cuckoo Hashing mit festen Blöcken vor.

3.4.2 Cache-Fehler

Wie zu Beginn dieses Kapitels erwähnt, trägt die Zahl der Cache-Fehler in erheblichem Maße zur Laufzeit eines Programms bei. Die neuen Verfahren wurden mit dem Ziel entwickelt, diese Anzahl klein zu halten. In einer Studienarbeit [Ott04] wurde experimentell untersucht, wie viele Cache-Fehler man bei den verschiedenen Verfahren pro Einfügeoperation im Schnitt erhält. Für verschiedene d und verschiedene ε sind die Diagramme für die gemittelten Cache-Fehler in Abbildung 3.13 auf der nächsten Seite angegeben.

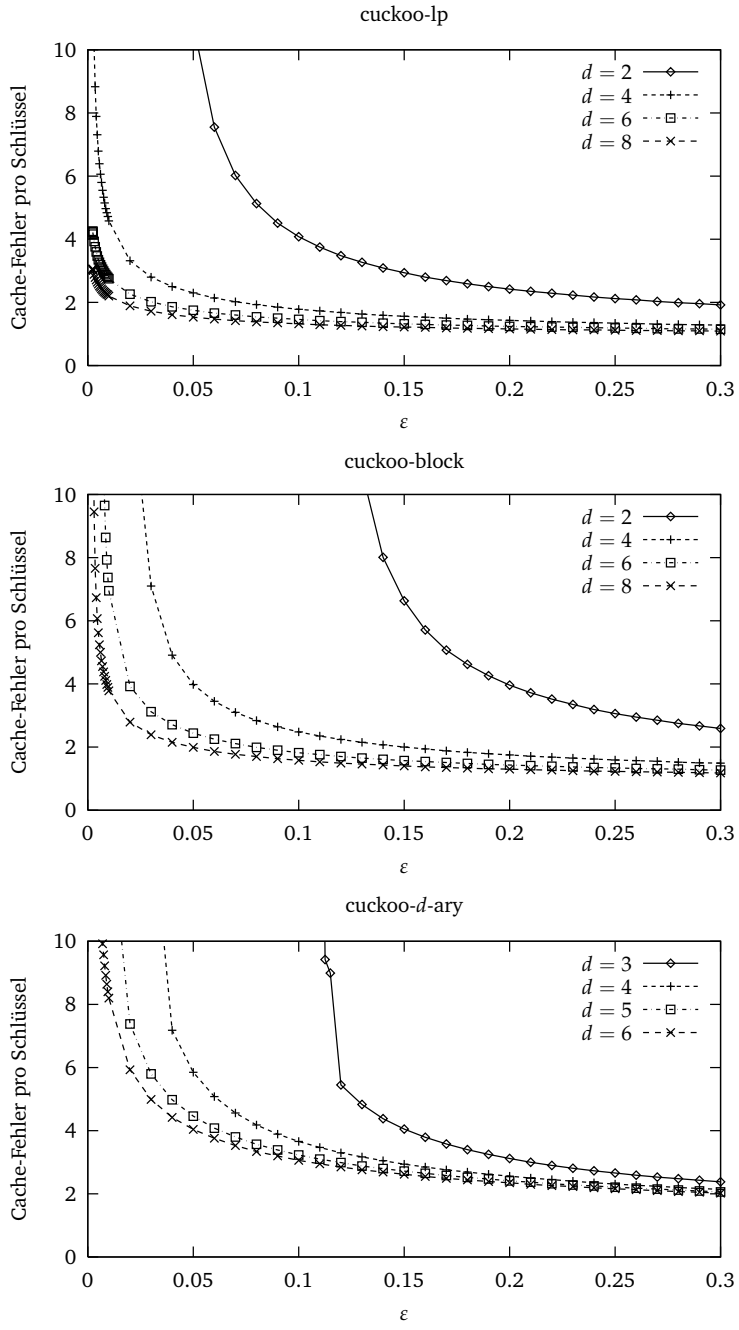
Dabei wurden $n = 10^6$ Schlüssel in ein anfangs leeres Wörterbuch mit $(1+\varepsilon)n$ Platz eingefügt. Die `insert`-Operation wurde ohne vorheriges Prüfen auf das Vorhandensein des einzufügenden Schlüssels implementiert. Als Hashklasse wurden Polynome dritten Grades verwendet.

Es wurde angenommen, dass jedes Mal, wenn während des Einfügens ein neuer Block beziehungsweise bei cuckoo-d-ary eine neue Zelle besucht wird, ein Cache-Fehler auftritt. Diese Annahme ist zwar sehr pessimistisch, jedoch kommt man bei wachsenden Schlüsselmengen diesem Verhalten immer näher.

Betrachtet man die Diagramme, sieht man, dass bei allen Verfahren die Zahl der Cache-Fehler sinkt, wenn d wächst. Damit man beispielsweise für $\varepsilon = 0.05$ bei cuckoo-d-ary mit nur 3 Cache-Fehlern auskommt, muss d ziemlich groß gewählt werden, auf jeden Fall größer als 6. Dabei muss man beachten, dass ein größeres d auch größere Laufzeiten einer `lookup`-Operation nach sich zieht. Bei den anderen Verfahren fallen große d weit weniger ins Gewicht. Dort kommt man für $d = 4$ bereits mit 2 Cache-Fehlern aus.

Diese Diagramme deuten an, dass sich mit den cachefreundlichen Verfahren für gegebenes ε und bei der Wahl eines geeigneten d schnellere Laufzeiten als mit cuckoo-d-ary erzielen lassen.

[5] Dieser Trick wurde uns von R. Pagh übermittelt.



Es wurde für jedes Verfahren die Zahl der durchschnittlichen Cache-Fehler pro Schlüssel beim Einfügen von 10^6 Schlüsseln in ein anfangs leeres Wörterbuch für verschiedene d in Abhängigkeit von ε gemessen.

Abbildung 3.13: Gemessene Zahl der Cache-Fehler bei `cuckoo-lp`, `cuckoo-block` und `cuckoo-`d`-ary` in Abhängigkeit von ε

3.4.3 Laufzeitmessungen

Um die Laufzeiten der verschiedenen Operationen zu ermitteln, wurden die Verfahren in der Sprache C++ implementiert und getestet. Es wurde folgendes Szenario verwendet: Die Schlüsselmenge S besteht aus $n = 10^6$ zufällige Zahlen aus dem Bereich $[2^{28}]$. Für $d = 2, \ldots, 32$ und verschiedene $m = (1+\varepsilon)n$ wurden die folgenden Experimente durchgeführt:

Exp. 1 Einfügen der n Schlüssel in ein anfangs leeres Wörterbuch mit m Zellen Platz. Hier interessiert neben der benötigten Zeit, ob alle Schlüssel eingefügt werden können.

Exp. 2 Alle n Schlüssel werden einmal gesucht. Dadurch erhält man die Zeit für eine positive Suche.

Exp. 3 Für eine Menge S' von n Schlüsseln, die garantiert nicht im Wörterbuch sind, wird jedes $x \in S'$ genau einmal gesucht. Mit diesem Experiment kann man die Zeit für eine negative Suche ermitteln.

Exp. 4 Es wird ein Schlüssel $x \in S$ gelöscht und dafür ein Schlüssel $x' \in S'$ eingefügt. Das heißt, in n Runden werden alle $x \in S$ aus T entfernt und dafür S' gespeichert. Durch dieses Experiment erhält man die Zeit für eine Lösch- und Einfügeoperation, wenn das Wörterbuch stark beladen ist.

Die benutzten Hashfunktionen $h_1, h_2, \ldots$ sind dabei Polynome dritten Grades:

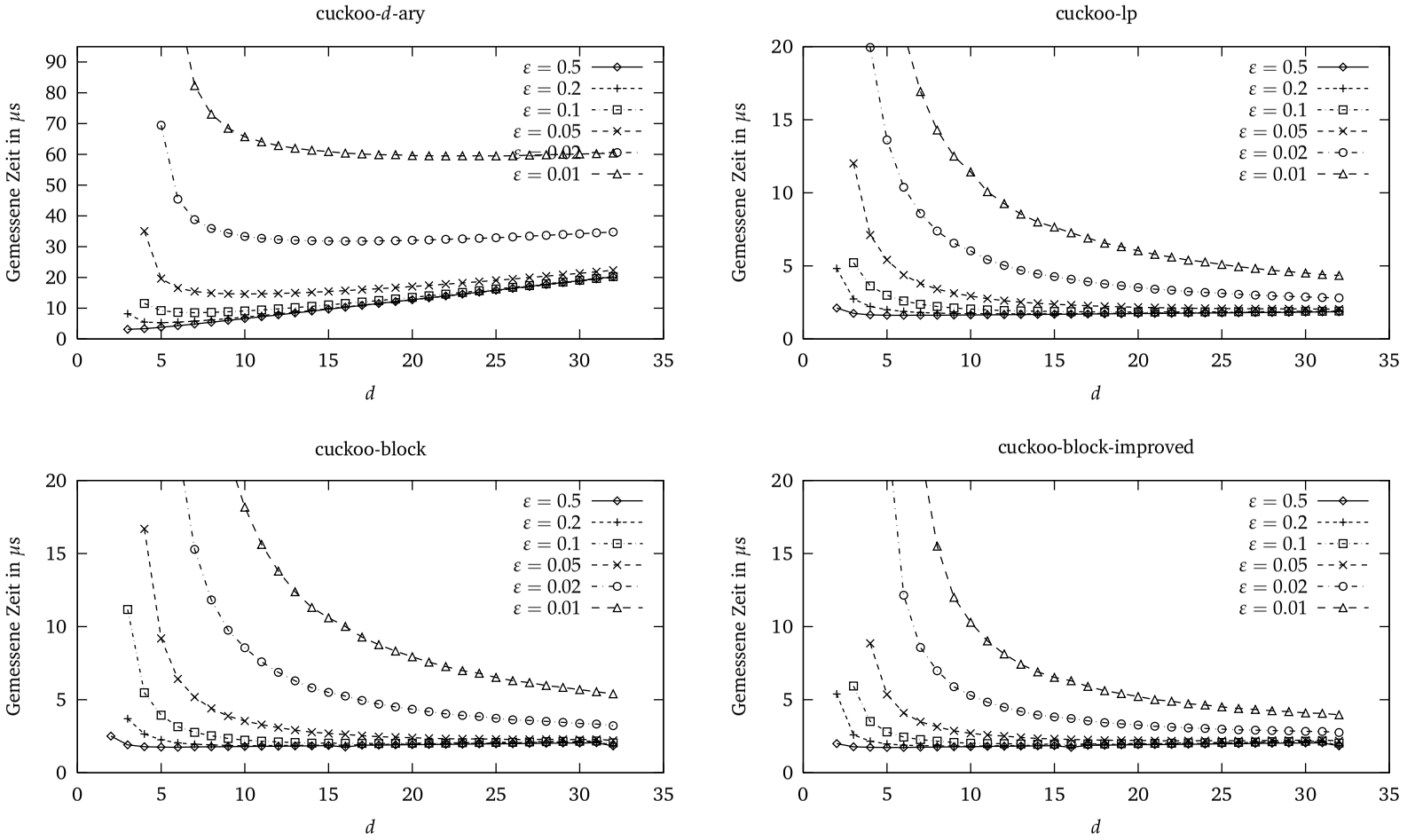
$$h_i \colon x \mapsto \sum_{j=0}^{3} a_j^{(i)} x^j \bmod p \bmod m \tag{3.118}$$

Die Zahl $p = 1073741827$ ist eine große Primzahl und die Koeffizienten sind zufällig gewählte Zahlen aus $[2^{31}]$.

Bei Verwendung fester Blöcke wurde darauf geachtet, dass in jedem Block alle im Block gespeicherten Schlüssel am Anfang des Blockes stehen. Das bedeutet einen geringfügig größeren Aufwand beim Löschen.

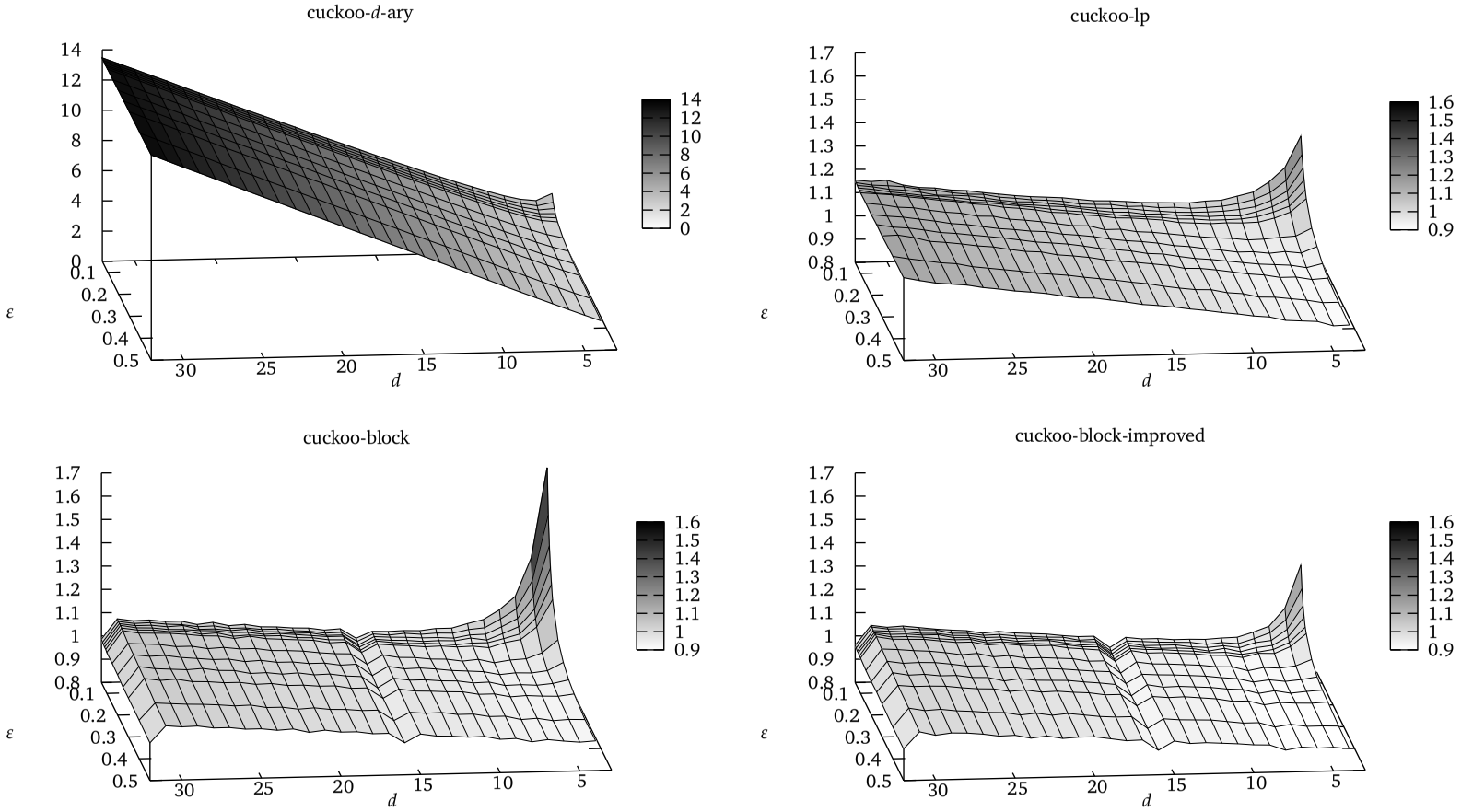
Alle Experimente wurden auf demselben Rechner[6] durchgeführt. Dabei wurde jeder Versuch 5 mal durchgeführt und die gemessenen Zeiten gemittelt.

[6]Prozessor: Intel(R) Pentium(R) 4 CPU 2.40 GHz stepping 07 (8 K L1 Cache, 512 K L2 Cache), Mainboard: GA-8PE800 i845PE ATX, RAM: DIMM DDR-333 512 MB, Umgebung: SuSE Linux 9.1 (Kernelversion 2.6.5), Compiler: Intel C Compiler für Linux Version 8.0 mit Optionen -O3 -xN -ipo



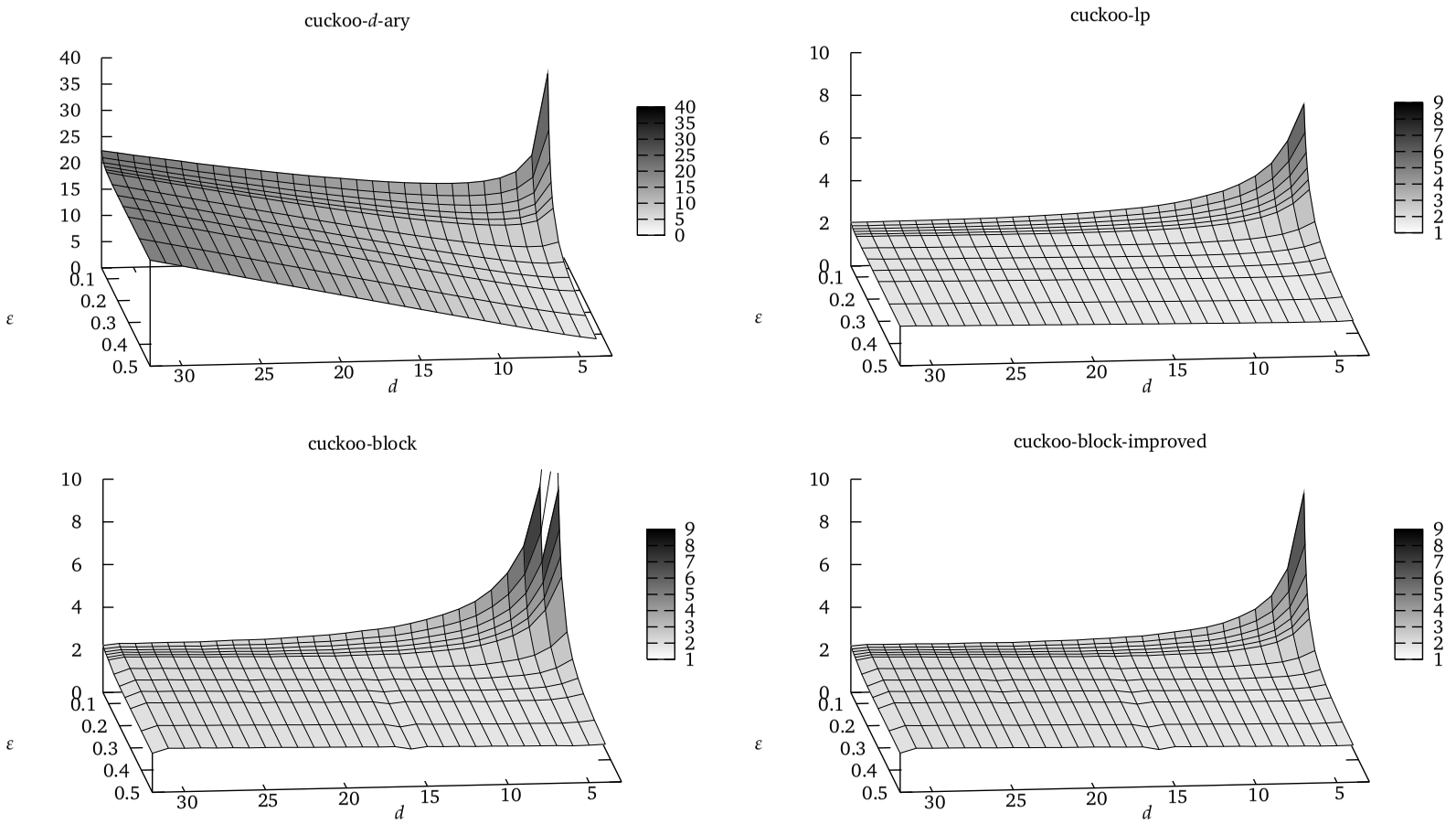
Für die Verfahren werden jeweils die gemittelten Laufzeit-Kurven für ein `delete` gefolgt von einem `insert` in Abhängigkeit von d (Abszisse) für verschiedene ε gezeigt. Die gemessenen Zeiten sind in μs angegeben (Ordinate).

Abbildung 3.14: Vergleich der Laufzeiten für jedes Verfahren für ein `delete` und ein `insert` bei gefülltem Wörterbuch



Für die Verfahren werden jeweils die gemittelten Laufzeit-Kurven für das Einfügen aller Schlüssel in ein anfangs leeres Wörterbuch für verschiedene ε und verschiedene d gezeigt. Die gemessenen Zeiten sind die durchschnittlichen Einfügezeiten pro Schlüssel in μs.

Abbildung 3.15: t-d-ε-Diagramm für jedes Verfahren für das Einfügen von n Schlüsseln in ein leeres Wörterbuch



Für die Verfahren werden jeweils die gemittelten Laufzeit-Kurven für ein `delete` gefolgt von einem `insert` für verschiedene ε und verschiedene d gezeigt. Die gemessenen Zeiten sind die durchschnittlichen Einfügezeiten pro Schlüssel in μs.

Abbildung 3.16: t-d-ε-Diagramm für jedes Verfahren für ein `delete` und ein `insert` bei gefülltem Wörterbuch

Wir betrachten zuerst für jedes Verfahren die gemittelten Zeiten für eine `insert`-Operation gefolgt von einer `delete`-Operation bei einem gefüllten Wörterbuch (Experiment 4) betrachtet. Man beachte dabei, dass man das d beim Verfahren cuckoo-d-ary eigentlich mit dem d der anderen Verfahren nicht in Beziehung setzen sollte.

Die Ergebnisse der Messungen sieht man in Abbildung 3.14 auf Seite 97. Um einen Überblick über die Laufzeiten bei verschiedenen Parametern zu bekommen, bieten sich dreidimensionale Diagramme an, welche die Zeit t in Abhängigkeit von den Parametern d und ε darstellen (Abbildungen 3.15 auf Seite 98 und 3.16 auf der vorherigen Seite). Im Folgenden heißen diese Diagramme *t-d-ε-Diagramme*. Obwohl man keine genauen Laufzeiten in diesen Diagrammen erkennt, sieht man doch sehr schön Trends und generelle Verhaltensweisen.

Generell fällt auf, dass die Laufzeit für größere ε besser ist als für kleinere. Das ist freilich keine Überraschung. Außerdem kann man an den Diagrammen ablesen, welches d man benötigt, um überhaupt mit $m = (1+\varepsilon)n$ Platz auszukommen.

Für große ε fällt außerdem auf, dass die Kurven bei cuckoo-d-ary mit wachsendem d stärker steigen als bei den anderen Verfahren. Das liegt daran, dass in der `insert`-Operation und in der `delete`-Operation jeweils eine Suche versteckt ist. Bei cuckoo-d-ary müssen d im Speicher verstreute Zellen untersucht werden und obendrein d Hashfunktionen ausgewertet werden, während bei den anderen Verfahren die zu untersuchenden $2d$ Zellen in zwei zusammenhängenden Blöcken liegen.

ε	cuckoo-d-ary		cuckoo-block		cuckoo-block-impr.		cuckoo-lp		lp
	Zeit	d	Zeit	d	Zeit	d	Zeit	d	Zeit
0.5	1.618	3	0.924	5	0.894	8	0.908	5	0.38
0.2	2.142	3	0.938	8	0.906	8	0.94	5	0.388
0.1	2.678	4	0.952	16	0.914	16	0.98	8	0.4
0.05	3.376	5	0.966	16	0.926	16	1.024	11	0.418
0.02	4.46	6	1.002	32	0.95	16	1.094	15	0.466
0.01	5.354	7	1.02	32	0.982	32	1.152	18	0.546

Tabelle 3.2: Kleinste durchschnittliche Dauer einer `insert`-Operation bei anfangs leerem Wörterbuch

Beim Betrachten der Bilder erkennt man bereits, dass bei festem ε die Laufzeit von cuckoo-d-ary größer ist als die der anderen Verfahren. Um dies genauer zu untersuchen, haben wir für jedes ε in jedem Verfahren das d gesucht, für das die Laufzeit am kleinsten ist, und zwar für jedes

ε	cuckoo-d-ary		cuckoo-block		cuckoo-block-impr.		cuckoo-lp		lp
	Zeit	d	Zeit	d	Zeit	d	Zeit	d	Zeit
0.5	0.846	3	0.708	2	0.794	2	0.668	14	0.354
0.2	0.838	3	0.76	3	0.788	2	0.682	12	0.36
0.1	0.834	3	0.766	3	0.806	3	0.69	24	0.368
0.05	1.048	4	0.79	4	0.814	5	0.702	23	0.384
0.02	1.24	5	0.802	5	0.82	6	0.72	23	0.416
0.01	1.24	5	0.81	6	0.82	6	0.732	28	0.462

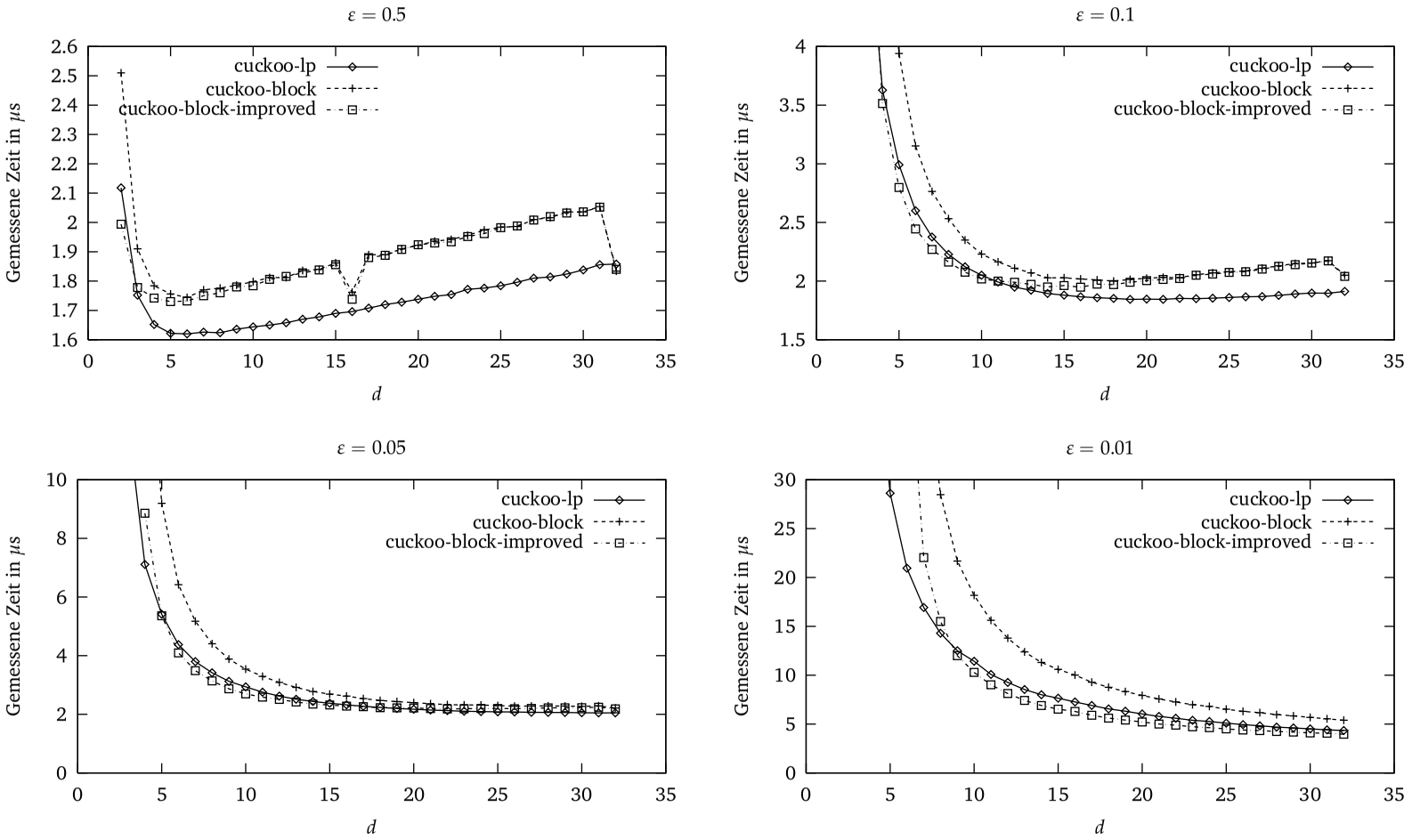
Tabelle 3.3: Kleinste durchschnittliche Dauer einer positiven `lookup`-Operation

ε	cuckoo-d-ary		cuckoo-block		cuckoo-block-impr.		cuckoo-lp		lp
	Zeit	d	Zeit	d	Zeit	d	Zeit	d	Zeit
0.5	1.286	3	0.814	2	0.798	2	0.816	2	0.378
0.2	1.278	3	0.82	3	0.794	2	0.806	2	0.43
0.1	1.272	3	0.82	3	0.798	3	0.814	3	0.548
0.05	1.678	4	0.826	4	0.8	4	0.81	3	0.934
0.02	2.086	5	0.834	5	0.812	5	0.81	3	3.012
0.01	2.092	5	0.834	6	0.822	6	0.808	3	10.114

Tabelle 3.4: Kleinste durchschnittliche Dauer einer negativen `lookup`-Operation

Experiment. Außerdem wurden diese Experimente auch für das Lineare Sondieren (*lp*) durchgeführt. Die Ergebnisse sieht man in den Tabellen 3.2 bis 3.5. Hier fällt auch auf, dass das größte $d = 32$ häufig das beste d ist; es kann durchaus sein, dass sich die Verfahren für größere d noch besser verhalten.

Bei lp gibt es das zu Beginn dieses Kapitels erwähnte Problem mit den als gelöscht markierten Zellen. Um dieses Problem zu vermeiden, haben wir die in [Knu82, Abschnitt 6.4] vorgeschlagene Implementierung der `delete`-Operation gewählt. Dabei ist zu beachten, dass man mit allerlei weiteren Tricks den Aufwand der `delete`-Operation reduzieren kann, so dass man die Zahlen für die Laufzeit beim Experiment 4 äußerst vorsichtig interpretieren muss. Der generelle Trend bleibt allerdings der gleiche: Wenn im Wörterbuch wenig freier Platz vorhanden oder das Wörterbuch sehr dynamisch ist, dann muss man bei lp für das Löschen, das Einfügen und für die negativen Lookups mit einer großen erwarteten Laufzeit rechnen. Hierzu sei noch einmal auf die erwartete Anzahl von besuchten Zellen bei den verschiedenen Operationen verwiesen (Abschnitt 3.1.1 auf Seite 54).



Für die Verfahren werden jeweils die gemittelten Laufzeit-Kurven für ein `delete` gefolgt von einem `insert` in Abhängigkeit von d (Abszisse) für verschiedene ε gezeigt. Die gemessenen Zeiten sind in μs angegeben (Ordinate).

Abbildung 3.17: Vergleich der Laufzeiten der cachefreundlichen Verfahren für verschiedene ε für ein `delete` und ein `insert` bei gefülltem Wörterbuch

ε	cuckoo-d-ary		cuckoo-block		cuckoo-block-impr.		cuckoo-lp		lp
	Zeit	d	Zeit	d	Zeit	d	Zeit	d	Zeit
0.5	3.124	3	1.744	6	1.73	5	1.62	6	2.184
0.2	5.296	5	1.844	10	1.822	10	1.718	12	5.894
0.1	8.54	7	2	18	1.948	16	1.844	19	16.3
0.05	14.614	10	2.214	32	2.182	22	2.048	31	54.82
0.02	31.804	17	3.222	32	2.754	32	2.808	32	297.45
0.01	59.478	22	5.398	32	3.964	32	4.344	32	1100.57

Tabelle 3.5: Kleinste durchschnittliche Dauer einer `delete`- gefolgt von einer `insert`-Operation bei gefülltem Wörterbuch

Vergleicht man die drei Verfahren cuckoo-lp, cuckoo-block und cuckoo-block-improved untereinander, stellt man fest, dass die Frage, ob das Verfahren mit variablen oder festen Blockgrenzen benutzt werden soll, nicht generell entschieden werden kann, insbesondere dann, wenn wenig Platz im Wörterbuch frei ist. In manchen Experimenten hat das eine Verfahren die Nase leicht vorn, in anderen Experimenten ist es umgekehrt. Entscheidet man sich für ein Verfahren mit festen Blockgrenzen, sollte man cuckoo-block-improved benutzen: Besonders, wenn das Feld bereits gefüllt ist, schneidet es nie schlechter als cuckoo-block ab.

Wenn noch genügend Platz ($\varepsilon = 0.5$) vorhanden ist, ist cuckoo-block-improved bei den Experimenten 2 und 4 langsamer als cuckoo-block. Das liegt daran, dass bei cuckoo-block-improved die Berechnung der Hashfunktionen aufwändiger ist. Weil beim Suchen meistens bereits nach dem Besuch weniger Zellen der Schlüssel gefunden wird und beim Einfügen kaum Zellinhalte getauscht werden müssen, werden die Resultate maßgeblich vom Aufwand für das Berechnen der Hashfunktion beeinflusst.

Kehren wir zuletzt noch einmal zu den Kurven der drei Verfahren cuckoo-lp, cuckoo-block und cuckoo-block-improved für das Experiment 4 in Abbildung 3.17 auf der vorherigen Seite zurück. Man sieht, dass für große d und große ε cuckoo-lp besser funktioniert. Die Ursache dafür ist die folgende: Die Löschoperation ist im wesentlichen eine positive Suche. Wenn viel Platz vorhanden ist ($\varepsilon = 0.5$), werden bei cuckoo-lp viele Schlüssel x genau auf $h_1(x)$ liegen, während die wenigsten Schlüssel bei den anderen beiden Verfahren in der ersten Zelle der Blockes $h_1(x)$ gespeichert sind. In einem solchen Fall muss nämlich der Block $h_2(x)$ nicht mehr betrachtet werden, was einen Cache-Fehler vermeidet. Für wachsendes d macht sich dieser Effekt immer stärker bemerkbar. Hinzu kommt, dass das Stopfen von Löchern während einer `delete`-Operation bei Verwendung fes-

d	cuckoo-lp	cuckoo-block	A	B
2	0.038394	0.115584	0.535156	0.7358
3	0.007117	0.043228	0.208261	0.5413
4	0.001975	0.02061	0.105351	0.3983
5	0.000724	0.01102	0.059569	0.2931
6	0.000332	0.006375	0.035834	0.2156
7	0.000178	0.003828	0.022396	0.1586
8	0.000113	0.002393	0.014370	0.1167
9	0.000076	0.001551	0.009402	0.0859
10	0.000062	0.001024	0.006234	0.0632
11	0.000048	0.000686	0.004176	0.0465

Die ersten beiden Spalten wurden mit $n \approx 2 \cdot 10^7$ Schlüsseln gemessen. Die Spalte A enthält das kleinste ε, welches (3.24) auf Seite 70 erfüllt. Die Spalte B enthält das kleinste ε, das Lemma 7 erfüllt.

Tabelle 3.6: Kleinstes ε bei gegebenem d

ter Blöcke mit größer werdendem d ebenfalls immer länger dauert.

Eine merkwürdige Beobachtung sei noch erwähnt: Die Zacken für $\varepsilon = 0.5$ in Abbildung 3.17 auf Seite 102. Offenbar kommt der Computer an dieser Stelle mit $d = 16$ und $d = 32$ sehr gut zurecht – oder mit den anderen d sehr schlecht. Über die genaue Ursache für dieses Phänomen können wir nur mutmaßen: Bei $d = 16$ und $d = 32$ sind alle Blöcke exakt an durch 64 teilbaren Speicheradressen ausgerichtet. Wenn die Größe eines Cache-Blockes (*cache line*) ebenfalls 64 ist, reicht bei der Untersuchung eines jeden Blockes immer genau ein Cache-Fehler aus. Wir sind uns aber nicht mit letzter Gewissheit darüber klar, ob das der alleinige Grund für das Phänomen ist; durch Wiederholen der Experimente auf einer anderen Rechnerplattform ließen sich sicher genauere Schlussfolgerungen ziehen.

3.4.4 Dichte

Zuletzt bleibt die Frage zu klären, wie viele Schlüssel man in das Feld packen kann, wenn d gegeben ist. Dazu wurden so lange Schlüssel in ein Feld mit $2 \cdot 10^7$ Zellen eingefügt, bis das Einfügen nicht mehr funktionierte. Für jeden Schlüssel durfte 10000 mal eine Hashfunktion ausgewertet werden. Dann wurde für jedes d der Wert für $1 + \varepsilon$ als Quotient aus der Anzahl der Zellen und der gespeicherten Schlüssel gebildet.

Das erhaltene ε ist in Tabelle 3.6 aufgelistet. Dabei sieht man unter an-

derem, dass das d von Lemma 7 auf Seite 67 wohl viel zu groß ist. [7] Außerdem sieht man, dass bei cuckoo-lp das Feld besser beladen werden kann als bei cuckoo-block, wenn man eine feste Anzahl von Cache-Fehlern pro Schlüssel festlegt.

3.5 Dynamisches Hashing mit Zeichenketten

Die gute Ausnutzung des Caches der hier neu vorgestellten Verfahren zahlt sich nur dann aus, falls die Schlüsselmenge S direkt im Feld T abgespeichert ist. Allerdings gibt es zahlreiche Anwendungen, bei denen das nicht erwünscht oder nicht möglich ist.

Möchte man beispielsweise Zeichenketten (*strings*) in T unterbringen, speichert man niemals die Zeichenketten selber in T, sondern Zeiger (*pointer*) auf die eigentlichen Speicherorte der Zeichenketten ab.[8] Die Zeichenketten selber können weit über den Hauptspeicher verstreut liegen, so dass der Vorteil der wenigen Cache-Fehler unserer Verfahren kaum noch eine Rolle spielt, sobald man die Zeichenketten lesen muss. Wenn auf eine Zeichenkette zugegriffen wird, sprechen wir im Folgenden von einem *Zeichenkettenzugriff*.

Dem kann man aber mit Fingerabdrücken (*finger prints*) einigermaßen beikommen: Ein Eintrag für einen Schlüssel x in T enthält neben der Adresse `p` von x im Speicher zusätzlich einen Fingerabdruck `fp` von x.

Zur Erläuterung dieser Technik wollen wir die Programmiersprache *C++*[9] benutzen.

In C++ könnte eine Datenstruktur `entry` für einen solchen Eintrag zum Beispiel wie folgt aussehen:

```
struct entry{
  int fp;
  char *p;
}
```

[7] Man muss hier Acht geben, denn konkrete n können täuschen. Es kann durchaus sein, dass für sehr viel größere n die gemessenen d sich ändern.

[8] Man beachte, dass solche Zeiger in verschiedenen Programmiersprachen verschiedene Gestalt besitzen. Wir wollen davon ausgehen, dass die Zeiger Adressen auf Speicherzellen sind.

[9] Weil die Verwendung dieser Technik sehr stark von den Basis-Datentypen *Ganzzahl* und *Zeiger* sowie von der Implementierung von Zeichenketten abhängt, erscheint uns die Verwendung von Pseudocode – im Gegensatz zu den vorangegangenen Abschnitten – an dieser Stelle nicht angebracht.

Will man nun zwei Zeichenketten auf Gleichheit testen, prüft man zunächst die in den entsprechenden Einträgen des Typs `entry` gespeicherten Fingerabdrücke und greift erst bei deren Gleichheit auf die eigentlichen Zeichenketten zu. Die folgende C++-Funktion `is_equal` implementiert diesen Algorithmus:

```
bool is_equal( const entry &a, const entry &b ){
  return a.fp != b.fp ? false : ( strcmp( a.p, b.p ) == 0 );
}
```

Wenn die Fingerabdrücke uniform und unabhängig über einem Bereich F verteilt sind, dann ist die Wahrscheinlichkeit, beim Vergleich zweier zufällig gewählter Zeichenketten auf die eigentlichen Adressen zugreifen zu müssen, $1/|F|$.

Wir wollen diese Technik auf das Verfahren mit festen Blöcken anwenden (Abschnitt 3.2.2). Als Fingerabdruck für x wollen wir $d \cdot (h_1(x) + h_2(x))$ wählen. Das bietet den Vorteil, dass man den Startindex im Feld T für den alternativen Block eines Schlüssels x', der in der Zelle j gespeichert ist, ganz billig ermitteln kann, nämlich mit $T[j].\texttt{fp} - i$, wobei i der Startindex des Blockes ist, in dem die Zelle j liegt, das heißt $i = j - (j \bmod d)$. Als Markierung `empty` für eine leere Zelle in T wollen wir $\texttt{empty.p} = \texttt{NULL}$ und $\texttt{empty.fp} = -1$ festlegen. Weil $h_1(x) + h_2(x) \geq 0$, ist sichergestellt, dass `is_equal` stets `false` zurückgibt, falls eine Zeichenkette mit dem Inhalt einer leeren Zelle verglichen wird.

Das Einfügen ist wie bei den Experimenten im letzten Abschnitt mit einem *random walk* realisiert. Damit sieht die Einfügeoperation `insert` wie folgt aus:[10]

```
bool insert( const char *px ){
  if( lookup( px ) )
    return false;

  int i = d*h1( px ), i1 = d*h2( px );
  entry x;
  x.p = px;
  x.fp = i1+i;
  if( rand() & 0x100 != 0 ) swap( i, i1 );

  while( --account > 0 ){
```

[10] Man sollte nach Möglichkeit den nicht sehr guten Zufallszahlengenarator `rand( )` durch einen besseren ersetzen, wie etwa `lrand48( )`.

```
    for( int j = i+d-1; j >= i; j-- ){
      if( is_empty( T[j] ) ){
        T[j] = x;
        return true;
      }
    }
    swap( x, T[i + rand( ) % d] );
    i = x.fp - i;
  }
  return false;
}
```

Man beachte, dass für das Einfügen eines Schlüssel x lediglich die beiden Hashwerte $h_1(x)$ und $h_2(x)$ berechnet werden müssen. Ansonsten werden die Hashfunktionen im weiteren Verlauf nicht mehr benötigt.

Das Suchen eines Schlüssels x funktioniert wie üblich. Es sei darauf hingewiesen, dass die Auswertung der beiden Hashfunktionen h_1 und h_2 parallel erfolgen kann, so dass die Zeichenkette nur einmal gelesen werden muss.

```
int lookup( const char *px ){
  int i1( d*h1( px ) ), i2( d*h2( px ) );
  entry x;
  x.p = px;
  x.fp = i1+i2;
  int _i1 = i1+d-1, _i2 = i2+d-1;
  for( ; _i1 >= i1; _i1--, _i2-- ){
    if( !is_empty( T[_i1] ) && is_equal( T[_i1], x )
      return _i1;
    if( !is_empty( T[_i2] ) && is_equal( T[_i2], x )
      return _i2;
  }
  return -1;
}
```

Dass bei der negativen Suche nach einem Schlüssel x die Wahrscheinlichkeit eines Zeichenkettenzugriffs (außer dem zur Berechnung von $h_1(x)$ und $h_2(x)$) sehr klein ist, kann man sich wie folgt überlegen: Es muss nur dann ein Zeichenkettenzugriff erfolgen, wenn es einen Schlüssel $x' \in S$ gibt, so

dass $\Gamma_B(x) = \Gamma_B(x')$ gilt. Dann erhalten wir:

$$\begin{aligned}\textbf{Prob}(\exists x' \in S\colon \Gamma_B(x') = \Gamma_B(x)) &\leq \textbf{Prob}(\exists x' \in S\colon \Gamma_B(x') \subseteq \Gamma_B(x)) \\ &\leq n\left(\frac{2}{m}\right)^2 = O(n^{-1}) \quad (3.119)\end{aligned}$$

Das heißt, dass bis auf eine verschwindend geringe Zahl von Fällen bei einer negativen Suche kein Zeichenkettenzugriff erfolgt. Im Fall einer positiven Suche kann man analog ermitteln, dass meistens ein Zeichenkettenzugriff ausreicht, der beweist, dass x in T gespeichert ist.

Um die Hashfunktionen h_1 und h_2 schnell auszuwerten, kann man eine Carter-Wegman-Klasse [CW79] benutzen: Zuerst wird eine große Tabelle `TABLE` mit Zufallszahlen gefüllt. Die Berechnung des Hashwertes für einen String, auf den der Zeiger `px` zeigt, funktioniert dann wie folgt:

```
unsigned int hash( const char *px ) const {
  unsigned int h=0, i=0;
  while( *px != '\0' )
    h ^= TABLE[ i += *px++ ];
  return h % m;
}
```

Wenn die Länge der längsten Zeichenkette L beträgt, muss `TABLE` $255L$ Einträge haben.

Allerdings ist die Berechnung eines Hashwertes mit dieser Funktion nicht cache-freundlich: Die Zugriffe auf die Tabelleneinträge sind nahezu willkürliche Speicherzugriffe.

Als Alternative zu einer großen Tabelle bietet sich ein Vorschlag aus [Sed02] an: Man benutzt einen primitiven Zufallszahlengenerator, um sich Zufallszahlen jedes Mal dann zu erzeugen, wenn man die Hashfunktion auswertet. Verschiedene Hashfunktionen unterscheiden sich dann lediglich im Startwert (*seed*) des Generators. Jedoch ist man dabei stark von der Qualität des Zufallsgenerators abhängig, insbesondere von dessen Periode und der Verteilung der Pseudozufallszahlen.

Eine weitere Möglichkeit besteht darin, für einen String $x = x_1 \ldots x_k$ k einfache Polynome $P_1, \cdots P_k$, $P_i(x) = (a_i x + b_i) \bmod p \bmod m$, für eine große Primzahl p und zufällig gewählte a_i, b_i, $1 \leq i \leq k$, auszuwerten und die Ergebnisse miteinander zu verknüpfen:

$$h(x) = (P_1(x_1) \otimes \cdots \otimes P_k(x_k)) \bmod m \qquad (3.120)$$

Dabei ist $\otimes$ entweder die Addition oder die Antivalenz (*xor*) für $U = 2^\ell$, $\ell \in \mathbb{N}$; für die Experimente wurde letztere gewählt. Außerdem wurden die

ε	Exp. 1				Exp. 2				Exp. 3				Exp. 4			
	string		string-smalltable		string		string-smalltable		string		string-smalltable		string		string-smalltable	
0.5	0.980	2	0.942	2	0.974	2	0.962	2	0.980	2	0.946	2	2.352	2	2.250	2
0.2	1.236	3	1.186	3	1.034	2	1.010	2	1.076	2	1.030	2	3.344	4	3.154	4
0.1	1.498	4	1.420	4	1.246	3	1.210	3	1.290	3	1.234	3	4.490	6	4.246	6
0.05	1.754	5	1.676	5	1.266	3	1.238	3	1.338	3	1.274	3	6.164	8	5.788	8
0.02	2.142	6	2.064	6	1.500	4	1.450	4	1.570	4	1.480	4	9.546	14	8.792	11
0.01	2.488	8	2.390	8	1.876	6	1.830	6	1.968	6	1.882	6	13.164	14	12.150	14

Es sind für jedes der Experimente auf Seite 96 die Laufzeiten in μs angegeben sowie die „besten“ d. Die Spalten mit dem Namen *string* enthalten die Werte bei Verwendung von Carter-Wegmann-Funktionen und die Spalten mit dem Namen *string-smalltable* enthalten die Werte bei Verwendung von einfachen Polynomen als Hashfunktionen.

Tabelle 3.7: Vergleich von Carter-Wegmann-Funktionen mit einfachen Polynomen

b_i nicht zufällig gewählt, sondern einfach $b_i = 0$ gesetzt. Zusätzlich wurde in den P_i auf alle Divisionen verzichtet, und erst, nachdem alle P_i verknüpft wurden, wurde durch p und durch m geteilt.

Vergleicht man die Laufzeiten für die beiden Varianten (Carter-Wegman und Polynome) der Hashfunktionen, sieht man, dass die Laufzeiten auch sehr ähnlich sind, so dass man ohne Verluste in der Laufzeit auf die Verwendung der Carter-Wegman-Klasse verzichten kann und dafür Speicherplatz und Cache-Fehler durch Verwendung von einfachen Polynomen sparen kann (siehe Tabelle 3.7).

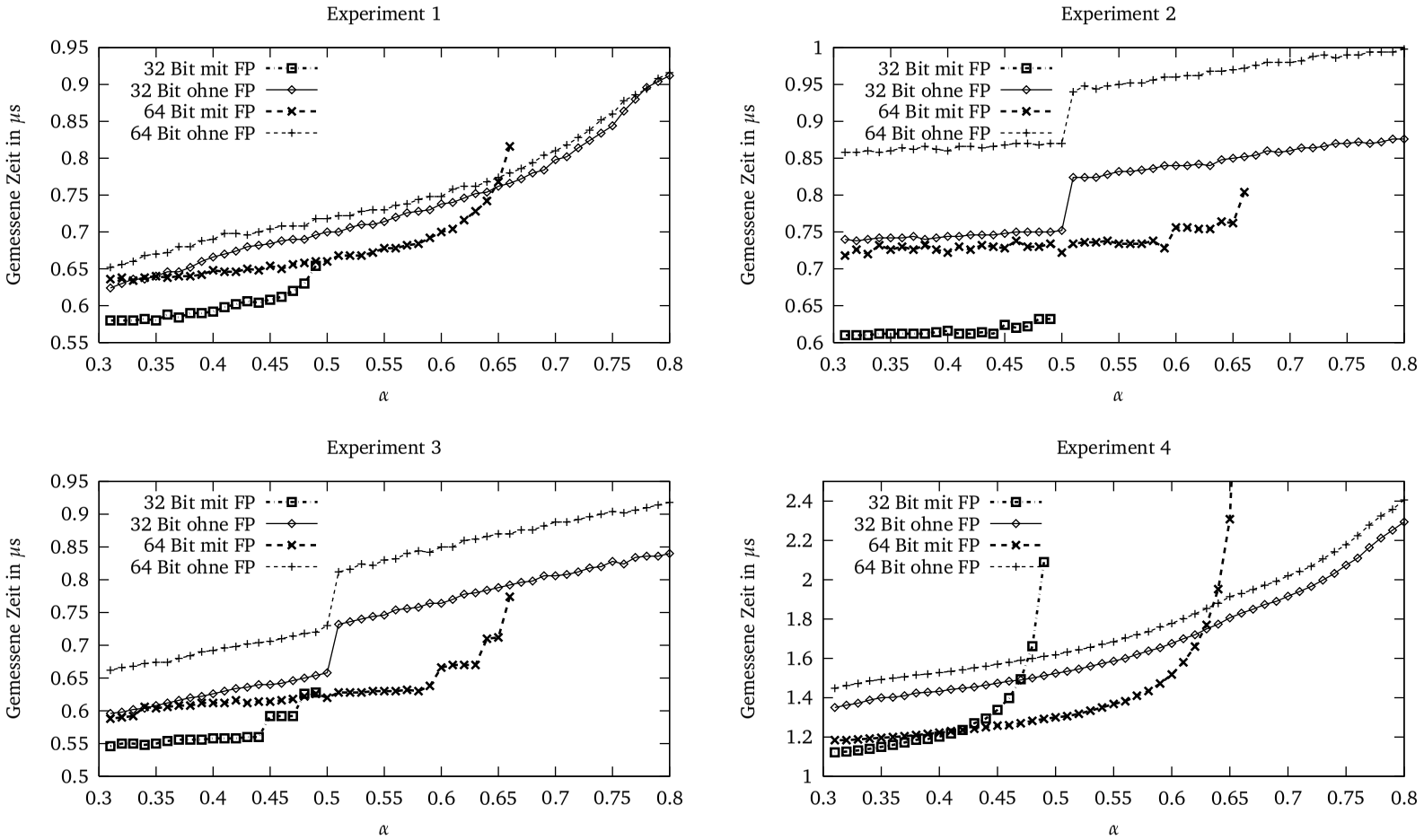
Um zu testen, ob durch Verwendung von Fingerabdrücken wirklich schnelle Laufzeiten für die verschiedenen Operationen erzielt werden können, wurden für 10^6 Zeichenketten die Experimente aus Abschnitt 3.4 auf Seite 93 wiederholt. Die Zeichenketten sind URLs aus einem Webcrawl [Dip01] und sind im Schnitt 57 Zeichen lang.

Um die Verfahren fair zu vergleichen, betrachten wir den *Speichernutzungsgrad*

$$\alpha := \frac{\text{Notwendiger Platz für } n \text{ Zeiger}}{\text{Verwendeter Platz}} = \frac{1}{1+\varepsilon} \cdot \frac{A}{A+B}. \tag{3.121}$$

Dabei ist A die Anzahl der Bytes, die ein Zeiger benötigt und B die Anzahl der Bytes, die ein Fingerabdruck benötigt. Wenn man keine Fingerabdrücke benutzt, ist $B = 0$.

Wir wollen mit $\alpha_{\max}$ die kleinste obere Schranke für den erreichbaren Speichernutzungsgrad bezeichnen. Man muss sich an dieser Stelle klar machen, dass diese Zahl bei Verwendung von Fingerabdrücken auf verschiedenen Systemen variiert, da A variiert. Auf einem 32-Bit-System ist $A = 4$,



Die gemessenen Zeiten sind über alle Schlüssel gemittelt und in μs (Ordinate) in Abhängigkeit vom Auslastungsgrad α angegeben. Abstufungen innerhalb einer Kurve (insbesondere bei Experiment 3) deuten auf unterschiedliche d hin. Man beachte, dass bei Verwendung von Fingerabdrücken auf einem 32-Bit-System $\alpha < 1/2$ und auf einem 64-Bit-System $\alpha < 2/3$ gilt.

Abbildung 3.18: Vergleich der Laufzeiten für *string-smalltable* mit und ohne Fingerabdrücken

32-Bit-Zeiger

ε		α	Exp. 1				Exp. 2				Exp. 3				Exp. 4			
mit FP	ohne FP		mit FP		ohne FP		mit FP		ohne FP		mit FP		ohne FP		mit FP		ohne FP	
0.32	1.63	0.38	0.590	3	0.652	1	0.612	2	0.740	1	0.556	2	0.620	1	1.186	5	1.424	2
0.25	1.50	0.40	0.592	3	0.666	1	0.616	2	0.744	1	0.558	2	0.626	1	1.202	5	1.432	2
0.19	1.38	0.42	0.602	8	0.674	1	0.612	2	0.746	1	0.558	2	0.634	1	1.236	5	1.448	2
0.14	1.27	0.44	0.604	8	0.682	2	0.612	2	0.746	1	0.560	2	0.640	1	1.294	7	1.464	2
0.09	1.17	0.46	0.612	8	0.688	2	0.620	3	0.750	1	0.592	3	0.642	1	1.398	8	1.484	2
0.04	1.08	0.48	0.630	8	0.690	2	0.632	4	0.750	1	0.626	4	0.650	1	1.662	13	1.500	2
0.00	1.00	0.50			0.700	2			0.752	1			0.658	1			1.524	2

64-Bit-Zeiger

ε		α	Exp. 1				Exp. 2				Exp. 3				Exp. 4			
mit FP	ohne FP		mit FP		ohne FP		mit FP		ohne FP		mit FP		ohne FP		mit FP		ohne FP	
0.75	1.63	0.38	0.640	2	0.680	1	0.732	2	0.866	1	0.608	2	0.684	1	1.210	2	1.516	2
0.59	1.38	0.42	0.646	2	0.698	2	0.726	2	0.866	1	0.616	2	0.698	1	1.236	2	1.542	2
0.45	1.17	0.46	0.650	2	0.708	2	0.738	2	0.870	1	0.616	2	0.710	1	1.260	3	1.580	2
0.33	1.00	0.50	0.660	3	0.718	2	0.722	2	0.870	1	0.620	2	0.730	1	1.300	4	1.618	2
0.23	0.85	0.54	0.672	3	0.730	2	0.738	2	0.948	2	0.630	2	0.822	2	1.350	4	1.672	2
0.15	0.72	0.58	0.684	4	0.744	2	0.738	2	0.956	2	0.630	2	0.844	2	1.434	6	1.736	2
0.08	0.61	0.62	0.716	5	0.762	2	0.754	3	0.962	2	0.670	3	0.860	2	1.660	9	1.826	2
0.01	0.52	0.66	0.816	9	0.780	2	0.804	6	0.972	2	0.774	6	0.870	2	3.698	16	1.930	3

Für die Experimente sind für ein 32- und ein 64-Bit-System jeweils für verschiedene α die gemittelten Laufzeiten pro Operation in μs sowie die „besten“ d angegeben. Links sind die dem jeweiligen α entsprechenden ε angegeben.

Tabelle 3.8: Vergleich der Laufzeiten für *string-smalltable* mit und ohne Fingerabdrücken

während auf einem 64-Bit-System $A = 8$ ist. Um bis zu 2^{31} Schlüssel zu speichern, reicht es, $B = 4$ zu wählen. Auf einem 32-Bit-System ist dann stets $\alpha_{\max} = 1/2$, während auf einem 64-Bit-System $\alpha_{\max} = 2/3$ gilt. Verzichtet man auf Fingerabdrücke, ist $\alpha_{\max} = 1$.

Die Experimente wurden auf einem 32-Bit- und einem 64-Bit-System durchgeführt.[11] Die Zeichenketten, die in Experiment 4 neu eingefügt wurden, sind die gleichen Zeichenketten jeweils von einem `@` angeführt. Für verschiedene α wurde für verschiedene d die jeweils beste Zeit ermittelt. Die Resultate der verschiedenen Experimente sind als Grafik angegeben (siehe Abbildung 3.18 auf Seite 110) sowie für ausgewählte α tabellarisch (siege Tabelle 3.8 auf der vorherigen Seite)

Die Experimente legen nahe, dass man bei gleicher Platzausnutzung durch Verwendung von Fingerabdrücken einen Laufzeitgewinn erzielt, sofern α genügend weit von $\alpha_{\max} = 1/2$ (bei dem 32-Bit-System) beziehungsweise $\alpha_{\max} = 2/3$ (bei dem 64-Bit-System) entfernt ist. Liegt hingegen α in der Nähe von $\alpha_{\max}$, kehrt sich die Situation um. Man muss sich allerdings stets vor Augen halten, dass der Schnittpunkt der Kurven für die Zeiten mit und ohne Fingerabdrücke sehr von der Konfiguration des zu Grunde liegenden Systems abhängt.

Mit der einfachen Technik der Fingerabdrücke gelingt es also, bei gleichbleibendem Platzbedarf die Laufzeit zu verbessern, sofern man genügend Platz zur Verfügung hat.

3.6 Fazit und offene Fragen

Wir haben gezeigt, wie man cachefreundliche dynamische Wörterbücher mit nahezu optimalem Platz bei garantierter konstanter Zugriffszeit implementieren kann. Dabei kann man einen neuen Schlüssel unter Beachtung der berechneten Schranken in erwarteter konstanter Zeit einfügen. In Experimenten zeigt sich, dass diese neuen Datenstrukturen auch praktikabel sind. Insbesondere in sehr dynamischen Szenarien sind diese Verfahren eine Alternative zum Linearen Sondieren.

Durch Verwendung von Fingerabdrücken ist es leicht möglich, Datensätze, die per Zeiger referenziert werden, cache- und laufzeiteffizient im Wörterbuch zu speichern. Obendrein kann man erreichen, dass beim Einfügen eines neuen Schlüssels mit hoher Wahrscheinlichkeit nur zwei Hashfunktionen berechnet werden müssen, das heißt, dass der Aufwand

[11] Prozessor: AMD Athlon64 3200 (2.8 MHz), Speicher: 1 GB DDR-RAM 400, Umgebung: Gentoo Linux x86_64, Kernel 2.6.9, Compiler: GNU C++ (g++), Optimierungen: -march=k8 -O3 -fprefetch-loop-arrays -fomit-frame-pointer

für das Auswerten einer Hashfunktion nicht mehr allzu sehr ins Gewicht fällt.

Interessant wären weitergehende Untersuchungen, die man in Studien- oder Diplomarbeiten durchführen könnte:

- (analytisch/experimentell) Wie robust sind die Verfahren gegenüber einfachen Hashfunktionen? Reichen einfache Hashfunktionen aus, um eine gute Auslastung bei moderatem d zu erhalten?
- (analytisch/experimentell) Wie kann man den *random walk*, der in den Experimenten implementiert wurde, analysieren?
- (experimentell) Man kann die Verfahren auch so implementieren, dass anstelle einer Tabelle der Größe m zwei Tabellen der Größe $m/2$ benutzt werden. Wie verhält es sich dann mit der Laufzeit und dem Platzbedarf?
- (experimentell) Ein genauer Vergleich mit anderen Wörterbuchverfahren [McG].
- (experimentell) Moderne Compiler (GNU Compiler Collection Version 3.4, Intel C Compiler Version 8) bieten Befehle, um ganz gezielt Speicherbereiche in den Prozessorcache zu laden, während der Prozessor weiterrechnen kann [The04]. (Bei den genannten Compilern heißt die entsprechende Anweisung `__builtin_prefetch`.) Um einen Schlüssel x zu suchen, kann man zum Beispiel wie folgt vorgehen:

  1. Berechne $y_1 := h_1(x)$.
  2. Lade die Zelle $T[y_1]$ in den Cache (ohne zu warten, bis die Daten geladen sind).
  3. Berechne $y_2 := h_2(x)$.
  4. Lade die Zelle $T[y_2]$ in den Cache (ohne zu warten, bis die Daten geladen sind).
  5. Suche x in $T[y_1], \ldots, T[y_1 \oplus (d-1)]$ und danach in $T[y_2], \ldots, T[y_2 \oplus (d-1)]$.

  Lohnt sich diese Vorgehensweise?
- (experimentell) Lohnt es sich, für das parallele Auswerten von Hashfunktionen die in modernen Prozessoren enthaltenen Multimediaerweiterungen (MMX, 3DNow!, SSE, Altivec, ...) zu benutzen?

Betrachtet man die Resultate der Experimente, sieht man, dass die nachgewiesenen Schranken für d und für die erwartete Laufzeit viel zu hoch sind; offen ist, inwieweit man bessere Schranken zeigen kann. Dies muss aber das Ziel weiterer Forschungen sein.

Für einzelne d, wie etwa $d = 2$ und $d = 3$, kann man genauer beobachten, wie die Blöcke gefüllt sein müssen und kann dadurch die Analysen verfeinern. Außerdem kann man versuchen, die Analyse von $r_\varepsilon(\alpha)$ aus Ungleichung (3.24) auf Seite 70 zu verbessern (siehe auch Abbildung 3.6 auf Seite 70).

Literaturverzeichnis

[ABKU00] AZAR, Y., A. Z. BRODER, A. R. KARLIN und E. UPFAL: *Balanced Allocations*. SIAM J. Comput., 29(1):180–200, 2000.

[BCSV00] BERENBRINK, P., A. CZUMAJ, A. STEGER und B. VÖCKING: *Balanced Allocations: The Heavily Loaded Case*. In: *Proceedings of the Thirty-Second ACM Symposium on Theory of Computing*, Seiten 745–754. ACM Press, 2000.

[BKS00] BOJESEN, J., J. KATAJAINEN und M. SPORK: *Performance engineering case study: Heap construction*. J. Exp. Algorithmics, 5:15, 2000.

[Bol85] BOLLOBÁS, B.: *Random graphs*. London-Orlando etc.: Academic Press (Harcourt Brace Jovanovich, Publishers). XVI, 447 p., 1985.

[CHM92] CZECH, Z. J., G. HAVAS und B. S. MAJEWSKI: *An optimal algorithm for generating minimal perfect hash functions*. Information Processing Letters, 43(5):257–264, 1992.

[CHM97] CZECH, Z. J., G. HAVAS und B. S. MAJEWSKI: *Perfect hashing*. Theoretical Computer Science, 182(1–2):1–143, 1997.

[CLRS01] CORMEN, T. H., C. E. LEISERSON, R. L. RIVEST und CL. STEIN: *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, Second Auflage, 2001.

[CRS03] CZUMAJ, A., CH. RILEY und CH. SCHEIDELER: *Perfectly balanced allocation*. In: *RANDOM-APPROX 2003*, Band 2764 der Reihe *LNCS*, Seiten 240–251. Springer, 2003.

[CW79] CARTER, J. L. und M. N. WEGMAN: *Universal classes of hash functions.* Journal of Computer and System Sciences, 18(2):143–154, 1979.

[DH01] DIETZFELBINGER, M. und T. HAGERUP: *Simple minimal perfect hashing in less space.* In: *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, Band 2161 der Reihe *Lecture Notes in Computer Science*, Seiten 109–120, Berlin, 2001. Springer-Verlag.

[Die] DIETZFELBINGER, M.: *Gespräche*.

[Dip01] DIPARTIMENTO DI SCIENZE DELL'INFORMAZIONE, UNIVERSITÀ DEGLI STUDI DI MILANO: *Webgraph*. `http://webgraph-data.dsi.unimi.it/`, 2001.

[DK03] DIETZFELBINGER, M. und M. KUNDE: *A case against using Stirling's formula (unless you really need it)*. BEATCS, 80:153–, Juni 2003. Technical Contributions.

[DW03] DIETZFELBINGER, M. und PH. WOELFEL: *Almost random graphs with simple hash functions*. In: *Proceedings of the Thirty-Fifth ACM Symposium on Theory of Computing*, Seiten 629–638. ACM Press, 2003.

[FPSS03] FOTAKIS, D., R. PAGH, P. SANDERS und P. SPIRAKIS: *Space efficient hash tables with worst case constant access time*. In: *STACS 2003. 20th annual symposium of theoretical aspects on computer science, Berlin, Germany, February 27 - March 1, 2003. Proceedings.*, Band 2607 der Reihe *Lecture Notes in Computer Science*, Seiten 271–282. Springer-Verlag, Berlin, 2003.

[GKP94] GRAHAM, R. L., D. E. KNUTH und O. PATASHNIK: *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[Hoc03] HOCKEL, K.: *Untersuchungen zu Verfahren für perfektes Hashing*. Diplomarbeit, Technische Universität Ilmenau, 2003.

[HR90] HAGERUP, T. und CH. RÜB: *A guided tour of Chernoff bounds*. Information Processing Letters, 33(6):305–308, Februar 1990.

[HT01] HAGERUP, T. und T. THOLEY: *Efficient minimal perfect hashing in nearly minimal space*. In: *STACS 2001. 18th annual symposium of theoretical aspects on computer science, Dresden, Germany, February 15-17, 2001. Proceedings.*, Band 2010 der Reihe *Lecture Notes in Computer Science*, Seiten 317–326. Springer-Verlag, Berlin, 2001.

[Knu82] KNUTH, D. E.: *The Art of Computer Programming*, Band 3 / Sorting and Searching. Addison-Wesley Publishing Company, 2. Auflage, 1982. ISBN 0-201-03803-X.

[McG] MCGEOCH, C. C.: *The fifth DIMACS challenge dictionaries*. `http://www.cs.amherst.edu/~ccm/challenge5/dicto/`.

[MR95] MOTWANI, R. und P. RAGHAVAN: *Randomized Algorithms*. Cambridge University Press, 1995.

[Ott04] OTT, A.: *Experimenteller Vergleich einiger neuer Verfahren zur Konstruktion von dynamischen Wörterbüchern mit fixer Zugriffszeit*. Studienarbeit, 2004.

[Pag99] PAGH, R.: *Hash and Displace: Efficient evaluation of minimal perfect hash functions*. In: *Proceedings of the 6th international Workshop on Algorithms and Data Structures (WADS '99)*, Band 1663 der Reihe *Lecture Notes in Computer Science*, Seiten 49–54. Springer-Verlag, Berlin, 1999.

[PR01] PAGH, R. und F. F. RODLER: *Cuckoo Hashing*. In: *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, Band 2161 der Reihe *Lecture Notes in Computer Science*, Seiten 121–133, Berlin, 2001. Springer-Verlag.

[PR04] PAGH, R. und F. F. RODLER: *Cuckoo Hashing*. J. Algorithms, 51:122–144, 2004.

[Prz98] PRZYBYLSKI, S.: *The Cache Memory Book*. Morgan Kaufmann, 2. Auflage, 1998. ISBN 0-123-22980-4.

[PSW96] PITTEL, B., J. SPENCER und N. WORMALD: *Sudden emergence of a giant k-core in a random graph*. J. Comb. Theory, Ser. B, 67(1):111–151, 1996.

[San99] SANDERS, P.: *Fast priority queues for cached memory*. In: *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation*, Band 1619 der Reihe *Lecture Notes in Computer Science*, Seiten 312–327. Springer-Verlag, 1999.

[San01] SANDERS, P.: *Reconciling simplicity and realism in parallel disk models*. In: *Proceedings of the Twelfth annual ACM-SIAM Symposium on Discrete Algorithms*, Seiten 67–76. Society for Industrial and Applied Mathematics, 2001.

[Sed02] SEDGEWICK, R.: *Algorithmen in C++*. Pearson Studium, Dritte Auflage, 2002.

[SEK00] SANDERS, P., S. EGNER und J. KORST: *Fast concurrent access to parallel disks*. In: *Proceedings of the Eleventh annual ACM-SIAM Symposium on Discrete Algorithms*, Seiten 849–858. Society for Industrial and Applied Mathematics, 2000.

[Sie89] SIEGEL, A.: *On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications*. In: *30th annual Symposium on Foundations of Computer Science, October 30–November 1, 1989, Research Triangle Park, North Carolina*, Seiten 20–25. IEEE Computer Society Press, 1989.

[SS90] SCHMIDT, J. P. und A. SIEGEL: *The spatial complexity of oblivious k-probe hash functions*. SIAM J. Comput., 19(5):775–786, 1990.

[The04] THE GNU PROJECT: *Using the GNU Compiler Collection (GCC)*. `http://gcc.gnu.org/onlinedocs/gcc/`, 2004.

[TY79] TARJAN, R. E. und ANDREW C. C. YAO: *Storing a sparse table*. Commun. ACM, 22(11):606–611, 1979.

[WC81] WEGMAN, M. N. und J. L. CARTER: *New classes and applications of hash functions*. Journal of Computer and System Sciences, 22(3):265–279, 1981.

[ÖP03] ÖSTLIN, A. und R. PAGH: *Uniform hashing in constant time and linear space*. In: *Proceedings of the Thirty-Fifth ACM Symposium on Theory of Computing*, Seiten 622–628. ACM Press, 2003.

Anhang A

Quelltexte der Experimente

Hier sind die Quellen für die Experimente mit dynamischen Wörterbüchern (siehe Abschnitt 3.4) angegeben.

A.1 keygen.h

In dieser Datei ist ein einfacher Generator für eindeutige Schlüssel implementiert. Die erzeugten Schlüssel sind aufsteigend sortiert. Dabei werden 1.3 mal so viele Schlüssel erzeugt wie benötigt werden und nach einer Sortierung von Duplikaten bereinigt.

```
#ifndef __KEYGEN_H
#define __KEYGEN_H

#include <vector>
#include <iostream>

/** Ein Generator für eindeutige Schlüssel */
template< class _KT >
class KeyGen {
public:
  typedef _KT KeyType;

protected:
  int n; //! Anzahl der zu erzeugenden Schlüssel

  /** Durchsucht das zunächst sortierte Feld
      nach doppelten Einträgen. Wenn ein doppelter
      Eintrag gefunden wird, dann wir er durch
      den nächsten verschiedenen Eintrag überschrieben.
      Gibt einen Iterator auf das Ende des Feldes
      mit paarweise verschiedenen Einträgen zurück.
  */
  typename vector<KeyType>::iterator
  remove_doubles( typename vector<KeyType>::iterator first,
                  typename vector<KeyType>::iterator last) {
    if( first == last )
      return last;
    stable_sort( first, last );
    typename vector<KeyType>::iterator eq( first );
    typename vector<KeyType>::iterator dest( first+1 );
    for( ; first != last; first++ ) {
      if( !( *first == *eq ) ) {
        if( dest != first )
          *dest = *first;
        dest++;
        eq = first;
      }
    }
    return dest;
  };

public:
  /** Erzeugt einen Generator für _n Schlüssel */
  KeyGen( int _n ) {
    n = _n;
  };

  /** Erzeugt n paarweise verschiedene Schlüssel */
  vector<KeyType> operator( )( ) {
    vector<KeyType> keys( 13 * n / 10 );
    typename vector<KeyType>::iterator it;
    for( it = keys.begin( ); it != keys.end( ); it++ )
      *it = lrand48( ) >> 4;
    it = remove_doubles( keys.begin( ), keys.end( ) );
    int k = it - keys.begin( );
    /* Test, ob genügend Schlüssel erzeugt worden sind */
    if( k < n ) {
      cerr << "Nicht genügend Schlüssel erzeugt!" << endl;
      exit( -1 );
    }
    keys.resize( n );
    return keys;
  }
};

#endif
```

A.2 dictionary-common.h

Hier sind für alle untersuchten Wörterbücher gemeinsam benutzte Strukturen, wie zum Beispiel verschiedene Hashfunktionen, definiert.

```
#ifndef __dictionary_common_h__
#define __dictionary_common_h__

using namespace std;

class int_empty_cell {
public:
  bool is_empty( const int &k ) const {
    return k == -1;
  };
  operator int() const {
    return -1;
  };
};

class pchar_empty_cell {
public:
  bool is_empty( const char* &k ) const {
    return k == (char *)0;
  };
  operator char*() const {
    return (char *)0;
  };
};


#define not_an_index (-1)
#define found (-2)

#include <stdlib.h>

#define my_rand lrand48

inline int binrand( ) {
  return my_rand( ) & 0x10 ? 1 : 0;
};

template< class K >
class HashFun {
protected:
  unsigned long m_nBuckets;
public:
  HashFun( const unsigned long nBuckets = 1 ) {
    m_nBuckets = nBuckets;
  };
  ~HashFun( ) {};
  typedef K Key;
  typedef long HashVal;
};

#ifdef __INTEL_COMPILER
typedef __int64 UINT64;
typedef __int64 INT64;
#else
typedef unsigned long long int UINT64;
typedef long long int INT64;
#endif


template< int deg = 2 >
class intStaticHashFun_poly : public HashFun<int> {
protected:
  const static UINT64 m_bigPrime = 1073741827;
  UINT64 m_a[deg+1];
public:
  intStaticHashFun_poly( const unsigned long nBuckets = 1 ):
  HashFun<int>( nBuckets ) {
    for( int j = deg; j >= 0; j-- )
      m_a[j] = my_rand( );
  }

  HashVal operator()( const Key &s ) const {
    UINT64 ll = m_a[deg] * s + m_a[deg-1];
    for( int j = deg-2; j >= 0; j-- )
      ll = ( ll % m_bigPrime ) * s + m_a[j];
    int result = (int)( ll % m_bigPrime );
    return result % m_nBuckets;
  }
};

template< int deg = 2 >
class intStaticHashFun_poly_offset :
  public intStaticHashFun_poly<deg> {
protected:
  int offset;
public:
  intStaticHashFun_poly_offset(
    const unsigned long nBuckets = 1,
    const unsigned int _offset = 0 ) :
  intStaticHashFun_poly<deg>( nBuckets ) {
    offset = _offset;
  }

  inline HashFun<int>::HashVal operator()(
    const HashFun<int>::Key &s ) const {
    int result = intStaticHashFun_poly<deg>::operator()(s);
    return result + offset;
  }
};


template <class K, class HF, class EMPTY>
class Dictionary {
protected:
  double epsilon;
  int m_nkeys;
  double f_grow;
  int m;
  K *mem;
  int account;

  /** Dieser Schlüssel hat nicht gepasst */
  K remaining_key;

  EMPTY __empty;
public:
  Dictionary( const double &_epsilon=0.1 ) {
    remaining_key = __empty;
    set_epsilon( _epsilon );
    set_account(0);
```

```
    m_nkeys = 0;
    m = 0;
    mem = NULL;
    set_grow( 2 );
  };

  void clear( ) {
    for( int j = this->m-1; j >= 0; j-- )
      this->mem[ j ] = this->__empty;
    this->m_nkeys = 0;
    this->remaining_key = this->__empty;
  }

  ~Dictionary( ) {
    if( mem != NULL )
      delete mem;
  }

  inline void set_epsilon( const double & _epsilon ) {
    epsilon = _epsilon;
  };

  double get_loadfactor( ) const {
    return double(m) / m_nkeys;
  };

  void set_grow( const double &_f_grow ) {
    f_grow = _f_grow;
  }

  void set_account( const int _account ) {
    account = _account;
  };

  int get_account( ) const {
    return account;
  };

  int get_nkeys( ) const {
    return m_nkeys;
  };

  double get_avg_probes( ) const {
    return 1/epsilon + 1;
  };

  double get_element_probes( ) const {
    return 2/epsilon + 1;
  };
};

#endif
```

A.3 cuckoo-common.h

Hier ist die für alle auf Cuckoo Hashing basierenden Wörterbücher gemeinsam benutzte Struktur definiert.

```
#ifndef __cuckoo_common_h__
#define __cuckoo_common_h__

#include "dictionary-common.h"
#include <algorithm>
#include <math.h>

template <class K, class HF, class EMPTY>
class CuckooDictionary : public Dictionary<K,HF,EMPTY> {
protected:
  int d;
  HF h[2];
  int insert_grow_account;
  double account_factor;

public:
  CuckooDictionary( int _d = 4, const double &_epsilon=0.1 ) :
  Dictionary<K,HF,EMPTY>( _epsilon ) {
    d = _d;
    set_epsilon( _epsilon );
    account_factor = 3;
  };

  void set_epsilon( const double & _epsilon ) {
    Dictionary<K,HF,EMPTY>::set_epsilon( _epsilon );
    insert_grow_account =
      (int)(account_factor/log(1+this->epsilon) );
  }

  void set_account_factor( const double & _account_factor ) {
    account_factor = _account_factor;
  };

  double get_avg_probes( ) const {
    double res = std::max( d+1., 0.2/log(1.+this->epsilon));
    return res;
  };

  double get_element_probes( ) const {
    double res = 2*std::max( d+1., 2/log(1.+this->epsilon));
    return res;
  };
};

#endif
```

A.4 cuckoo-block.h

In dieser Datei ist das Cuckoo Hashing mit festen Blöcken (*cuckoo-block*) definiert.

```
#ifndef __cuckoo_block_h__
#define __cuckoo_block_h__

#include "cuckoo-common.h"

template <class K, class HF, class EMPTY>
class CuckooBlock : public CuckooDictionary<K,HF,EMPTY> {
protected:
  int d1;
  HF h[2];
```

```
public:
  CuckooBlock( int _d = 4, const double &_epsilon=0.1 ) :
  CuckooDictionary<K,HF,EMPTY>( _d, _epsilon ) {
    this->d1 = _d-1;
    reserve( 4*_d );
  };

  int insert( const K &_x ) {
    K x( _x );
    int freecell = not_an_index;

    int r = binrand();
    int i0 = h[r]( x )*this->d;
    int i1 = h[1-r]( x )*this->d;
    int i = i0;
    int ende = i0+this->d;

    for( ; i0 < ende; i0++, i1++ ) {
      if( this->mem[i0] == x )
        return found;
      else
        if( freecell == not_an_index &&
            this->__empty.is_empty(this->mem[i0]) )
          freecell = i0;
      if( this->mem[i1] == x )
        return found;
      else
        if( freecell == not_an_index &&
            this->__empty.is_empty(this->mem[i1]) )
          freecell = i1;
      if( this->__empty.is_empty(this->mem[i0]) &&
          this->__empty.is_empty(this->mem[i1]) )
        break;
    }

    this->account -= 2;

    if( freecell != not_an_index ) {
      this->mem[freecell] = x;
      this->m_nkeys++;
      return freecell;
    }

    while( this->account > 0 ) {
      int j = i + my_rand( ) % this->d;
      ::swap( x, this->mem[j] );
      r = binrand();
      i0 = h[r]( x )*this->d;
      i = i0;
      int ende = i0+this->d;
      this->account--;
      for( ; i0 < ende; i0++ ) {
        if(this->__empty.is_empty(this->mem[i0]) ){
          this->mem[i0] = x;
          this->m_nkeys++;
          return i0;
        }
      }
    }
    this->remaining_key = x;
    return not_an_index;
  }

  void remove
    ( const K & x ) {
    int i0 = h[0]( x ) * this->d;
    int i1 = h[1]( x ) * this->d;
    int block_ende0 = i0+this->d;
    int block_ende1 = i1+this->d;
    int ende, zelle = not_an_index;
    for( ; i0 < block_ende0; i0++, i1++ ) {
      if( this->mem[i0] == x ) {
        ende = block_ende0;
        zelle = i0;
        break;
      }
      if( this->mem[i1] == x ) {
        ende = block_ende1;
        zelle = i1;
        break;
      }
    }
    if( zelle == not_an_index )
      return;
    int _zelle = zelle;
    zelle++;
    while( zelle < ende &&
           !this->__empty.is_empty(this->mem[zelle]) )
      zelle++;
    zelle--; // Zelle zeigt auf den letzten belegten Index

    if( zelle > _zelle )
      this->mem[_zelle] = this->mem[zelle];
    this->mem[zelle] = this->__empty;
    this->m_nkeys--;
  }

  int insert_grow( const K & _x ) {
    this->account = this->insert_grow_account;
    int index = insert( _x );
    if( index != not_an_index )
      return index;
    if( !grow_by( this->f_grow ) )
      return not_an_index;
    return insert( this->remaining_key );
  };

  void clear( ) {
    for( int j = this->m-1; j >= 0; j-- )
      this->mem[ j ] = this->__empty;
    this->m_nkeys = 0;
    this->remaining_key = this->__empty;
  }

  /** Versucht, ncells Zellen zu reservieren und
  die Schlüssel dort unterzubringen. Wenn das
  nicht geht, wird false zurückgegeben. */
  bool reserve( int ncells ) {
    HF _h0 = h[0], _h1 = h[1];
    int oldm = this->m;
    int oldnkeys = this->m_nkeys;
    K old_remaining_key = this->remaining_key;
    int old_account = this->account;

    int nBlocks = (ncells+this->d1) / this->d;
    h[0] = HF( nBlocks );
    h[1] = HF( nBlocks );
    ncells = nBlocks * this->d;
```

```
    K *oldmem = this->mem;
    this->mem = new K[ ncells ];
    this->m = ncells;
    this->set_account( (int)(this->account_factor * this->m) );
    this->clear( );
    this->remaining_key = old_remaining_key;
    if( oldmem != NULL ) {
      for( int j = oldm-1; j >= 0; j-- ) {
        if( !this->__empty.is_empty(oldmem[j]) ) {
          if( insert(oldmem[j]) == not_an_index ) {
            h[0] = _h0;
            h[1] = _h1;
            delete[ ] this->mem;
            this->mem = oldmem;
            this->m = oldm;
            this->m_nkeys = oldnkeys;
            this->remaining_key = old_remaining_key;
            this->account = old_account;
            return false;
          }
        }
      }
      delete[ ] oldmem;
    }
    return true;
  };

  bool grow_by( const double &factor ) {
    int _m = (int)ceil( factor * this->m );
    if( factor > 1 && _m == this->m )
      _m++;
    return reserve( _m );
  }

  inline int lookup( const K &x ) const {
    int i0 = h[0]( x ) * this->d;
    if( this->mem[i0] == x )
      return i0;
    int i1 = h[1]( x ) * this->d;
    if( this->mem[i1] == x )
      return i1;
    int ende=i0+this->d;
    i0++;
    i1++;
    for( ; i0 < ende; i0++, i1++ ) {
      if( this->mem[i0] == x )
        return i0;
      if( this->mem[i1] == x )
        return i1;
    }
    return not_an_index;
  }
};

#endif
```

A.5 cuckoo-block-simple.h

In dieser Datei ist das Cuckoo Hashing mit festen Blöcken und verbesserten Hashfuntionen (*cuckoo-block-improved*) definiert.

```
#ifndef __cuckoo_block_simple_h__
#define __cuckoo_block_simple_h__

#include "cuckoo-common.h"

template< class K, class HF >
class HashFunDouble : public HashFun<K> {
protected:
  HF m_q, m_h;
public:
  HashFunDouble( const unsigned long nBuckets = 1 ) :
  HashFun<int>( nBuckets ), m_q(nBuckets), m_h(nBuckets ){}

  /** Berechnet q(x)-i */
  inline typename HashFunDouble<K,HF>::HashVal
  q( const typename HashFunDouble<K,HF>::Key &s,
     const typename HashFunDouble<K,HF>::HashVal &i ) const{
    int res = m_q(s) - i;
    return ( res < 0 ) ? this->m_nBuckets+res : res;
  }

  /** Berechnet h1(x), h2(x) */
  inline pair<typename HashFunDouble<K,HF>::HashVal,
  typename HashFunDouble<K,HF>::HashVal> operator()(
    const typename HashFunDouble<K,HF>::Key &s ) const{
    int _h = m_h(s), _q = m_q(s);
    int r1 = _q - _h;
    if( r1 < 0 )
      r1 += this->m_nBuckets;
    return make_pair( _h, r1 );
  }
};



template< int deg = 2 >
class intStaticHashFun_poly_double :
  public HashFun<int> {
protected:
  const static INT64 m_bigPrime = 1073741827;
  INT64 m_h[deg+1];
  INT64 m_q[deg+1];
public:
  intStaticHashFun_poly_double(
    const unsigned long nBuckets = 1 ) :
  HashFun<int>( nBuckets ) {
    for( int j = deg; j >= 0; j-- ) {
      m_q[j] = my_rand( );
      m_h[j] = my_rand( );
    }
  }

  /** Berechnet q(x)-i */
  HashVal q( const Key &s, const HashVal &i ) const {
    INT64 ll = m_q[deg] * s + m_q[deg-1];
    for( int j = deg-2; j >= 0; j-- )
```

```
      ll = ( ll % m_bigPrime ) * s + m_q[j];
    int result = (int)( ll % m_bigPrime );
    return ( result - i ) % this->m_nBuckets;
  }

  /** Berechnet h1(x), h2(x) */
  pair<HashVal,HashVal> operator()( const Key &s ) const{
    INT64 ll0 = m_h[deg] * s + m_h[deg-1];
    INT64 ll1 = m_q[deg] * s + m_q[deg-1];
    for( int j = deg-2; j >= 0; j-- ) {
      ll0 = ( ll0 % m_bigPrime ) * s + m_h[j];
      ll1 = ( ll1 % m_bigPrime ) * s + m_q[j];
    }
    int r0 = (int)( ll0 % m_bigPrime );
    int r1 = (int)( ll1 % m_bigPrime );
    return make_pair(
            r0 % this->m_nBuckets,
            (r1 - r0) % this->m_nBuckets );
  }
};


template<class K, class HF, class EMPTY>
class CuckooBlockSimple :
  public CuckooDictionary<K, HF, EMPTY> {
protected:
  int d1;
  HF h;
  int nBlocks;

public:
  CuckooBlockSimple( int _d=4, const double &_epsilon=0.1 ):
  CuckooDictionary<K,HF,EMPTY>( _d, _epsilon ) {
    d1 = _d-1;
    reserve( 4*this->d );
  };

  int insert( const K &_x ) {
    K x( _x );
    int freecell = not_an_index;

    pair<int,int> p( h(x) );
    int r = binrand();
    int ind[2] = {p.first, p.second};
    int _i;
    int i0 = (_i = ind[r]) * this->d;
    int i1 = ind[1-r] * this->d;
    int i = i0;
    int ende = i0+this->d;

    for( ; i0 < ende; i0++, i1++ ) {
      if( this->mem[i0] == x )
        return found;
      else
        if( freecell == not_an_index &&
            this->__empty.is_empty(this->mem[i0]) )
          freecell = i0;
      if( this->mem[i1] == x )
        return found;
      else
        if( freecell == not_an_index &&
            this->__empty.is_empty(this->mem[i1]) )
          freecell = i1;
      if( this->__empty.is_empty(this->mem[i0]) &&
          this->__empty.is_empty(this->mem[i1]) )
        break;
    }

    this->account -= 2;

    if( freecell != not_an_index ) {
      this->mem[freecell] = x;
      this->m_nkeys++;
      return freecell;
    }

    while( this->account > 0 ) {
      int j = i + my_rand( ) % this->d;
      swap( x, this->mem[j] );
      r = binrand();
      _i = h.q( x, _i );
      i0 = _i * this->d;
      i = i0;
      int ende = i0+this->d;
      this->account--;
      for( ; i0 < ende; i0++ ) {
        if( this->__empty.is_empty(this->mem[i0]) ) {
          this->mem[i0] = x;
          this->m_nkeys++;
          return i0;
        }
      }
    }
    this->remaining_key = x;
    return not_an_index;
  }

  int insert_grow( const K & _x ) {
    this->account = this->insert_grow_account;
    int index = insert( _x );
    if( index != not_an_index )
      return index;
    if( !grow_by( this->f_grow ) )
      return not_an_index;
    return insert( this->remaining_key );
  };

  void clear( ) {
    for( int j = this->m-1; j >= 0; j-- )
      this->mem[ j ] = this->__empty;
    this->m_nkeys = 0;
    this->remaining_key = this->__empty;
  }

  /** Versucht, ncells Zellen zu reservieren
  und die Schlüssel dort unterzubringen. Wenn
  das nicht geht, wird false zurückgegeben. */
  bool reserve( int ncells ) {
    HF _h = h;
    int oldm = this->m;
    int oldnkeys = this->m_nkeys;
    K old_remaining_key = this->remaining_key;
    int old_account = this->account;
    int old_nBlocks = nBlocks;

    nBlocks = (ncells+d1) / this->d;
    h = HF( nBlocks );
    ncells = nBlocks * this->d;

    K *oldmem = this->mem;
```

```
    this->mem = new K[ ncells ];
    this->m = ncells;
    this->set_account( (int)(this->account_factor * this->m) );
    this->clear( );
    this->remaining_key = old_remaining_key;
    if( oldmem != NULL ) {
      for( int j = oldm-1; j >= 0; j-- ) {
        if( !this->__empty.is_empty(oldmem[j]) ) {
          if( insert(oldmem[j]) == not_an_index ) {
            h = _h;
            delete[ ] this->mem;
            this->mem = oldmem;
            this->m = oldm;
            this->m_nkeys = oldnkeys;
            this->remaining_key = old_remaining_key;
            this->account = old_account;
            return false;
          }
        }
      }
      delete[ ] oldmem;
    }
    return true;
  };

  bool grow_by( const double &factor ) {
    int _m = (int)ceil( factor * this->m );
    if( factor > 1 && _m == this->m )
      _m++;
    return reserve( _m );
  }

  inline int lookup( const K &x ) const {
    pair<int,int> p( h(x) );
    p.first *= this->d;
    if( this->mem[p.first] == x )
      return p.first;
    p.first++;
    p.second *= this->d;
    int ende=p.first+this->d;
    if( this->mem[p.second] == x )
      return p.second;
    p.second++;
    for( ; p.first < ende; p.first++, p.second++ ) {
      if( this->mem[p.first] == x )
        return p.first;
      if( this->mem[p.second] == x )
        return p.second;
    }
    return not_an_index;
  }

  void remove( const K & x ) {
    pair<int,int> p( h(x) );
    p.first *= this->d;
    p.second *= this->d;
    int block_ende0 = p.first+this->d;
    int block_ende1 = p.second+this->d;
    int ende, zelle = not_an_index;
    for( ; p.first < block_ende0; p.first++, p.second++ ){
      if( this->mem[p.first] == x ) {
        ende = block_ende0;
        zelle = p.first;
        break;
      }
      if( this->mem[p.second] == x ) {
        ende = block_ende1;
        zelle = p.second;
        break;
      }
    }
    if( zelle == not_an_index )
      return;
    int _zelle = zelle;
    zelle++;
    while( zelle < ende &&
           !this->__empty.is_empty(this->mem[zelle]) )
      zelle++;
    zelle--; // Zelle zeigt auf den letzten belegten Index

    if( zelle > _zelle )
      this->mem[_zelle] = this->mem[zelle];
    this->mem[zelle] = this->__empty;
    this->m_nkeys--;
  }
};

#endif
```

A.6 cuckoo-lp.h

In dieser Datei ist das Cuckoo Hashing mit begrenzter Sondierungsweite (*cuckoo-lp*) definiert.

```
#ifndef __cuckoo_lp_h__
#define __cuckoo_lp_h__

#include "cuckoo-common.h"

template <class K, class HF, class EMPTY>
class CuckooLP : public CuckooDictionary<K,HF,EMPTY>{
protected:
  int d1;
  HF h[2];

public:
  CuckooLP( int _d = 4, const double &_epsilon=0.1 ) :
  CuckooDictionary<K,HF,EMPTY>( _d, _epsilon ) {
    d1 = _d-1;
    reserve( 4*this->d );
  };

  int insert( const K &_x ) {
    K x( _x );
    int freecell = not_an_index;

    int r = binrand();
    int i0 = h[r]( x );
    int i1 = h[1-r]( x );

    int i = i0;
    int ende = i0+this->d;

    for( ; i0 < ende; i0++, i1++ ) {
```

```
    if( this->mem[i0] == x )
      return found;
    else
      if( freecell == not_an_index &&
          this->__empty.is_empty(this->mem[i0]) )
        freecell = i0;
    if( this->mem[i1] == x )
      return found;
    else
      if( freecell == not_an_index &&
          this->__empty.is_empty(this->mem[i1]) )
        freecell = i1;
  }

  this->account -= 2;

  if( freecell != not_an_index ) {
    this->mem[freecell] = x;
    this->m_nkeys++;
    return freecell;
  }

  while( this->account > 0 ) {
    int j = i + my_rand( ) % this->d;
    swap( x, this->mem[j] );
    r = binrand();
    i0 = h[r]( x );
    i = i0;
    ende = i0+this->d;
    this->account--;
    for( ; i0 < ende; i0++ ) {
      if( this->__empty.is_empty(this->mem[i0]) ) {
        this->mem[i0] = x;
        this->m_nkeys++;
        return i0;
      }
    }
  }
  this->remaining_key = x;
  return not_an_index;
}

void remove
  ( const K & _x ) {
  int index = lookup( _x );
  if( index != not_an_index ) {
    this->mem[index] = this->__empty;
    this->m_nkeys--;
  }
}

int insert_grow( const K & _x ) {
  this->account = this->insert_grow_account;
  int index = insert( _x );
  if( index != not_an_index )
    return index;
  if( !grow_by( this->f_grow ) )
    return not_an_index;
  return insert( this->remaining_key );
};

void clear( ) {
  for( int j = this->m+d1-1; j >= this->m+d1; j-- )
    this->mem[j] = this->__empty;
  CuckooDictionary<K,HF,EMPTY>::clear( );
}

/** Versucht, ncells Zellen zu reservieren
und die Schlüssel dort unterzubringen. Wenn
das nicht geht, wird false zurückgegeben. */
bool reserve( int ncells ) {
  HF _h0 = h[0], _h1 = h[1];
  int oldm = this->m;
  int oldnkeys = this->m_nkeys;
  K old_remaining_key = this->remaining_key;
  int old_account = this->account;

  h[0] = HF( ncells );
  h[1] = HF( ncells );
  K *oldmem = this->mem;
  this->mem = new K[ ncells+d1 ];
  this->m = ncells;
  this->set_account( (int)(this->account_factor * this->m) );
  clear( );
  this->remaining_key = old_remaining_key;
  if( oldmem != NULL ) {
    for( int j = oldm+d1-1; j >= 0; j-- ) {
      if( !this->__empty.is_empty(oldmem[j]) ) {
        if( insert(oldmem[j]) == not_an_index ){
          h[0] = _h0;
          h[1] = _h1;
          this->m = oldm;
          this->m_nkeys = oldnkeys;
          this->remaining_key = old_remaining_key;
          this->account = old_account;
          delete[ ] this->mem;
          this->mem = oldmem;
          return false;
        }
      }
    }
    delete[ ] oldmem;
  }
  return true;
};

bool grow_by( const double &factor ) {
  int _m = (int)ceil( factor * this->m );
  if( factor > 1 && _m == this->m )
    _m++;
  return reserve( _m );
}

inline int lookup( const K &x ) const {
  int i0 = h[0]( x );
  if( this->mem[i0] == x )
    return i0;
  int i1 = h[1]( x );
  if( this->mem[i1] == x )
    return i1;
  int ende=i0+this->d;
  i0++;
  i1++;
  for( ; i0 < ende; i0++, i1++ ) {
    if( this->mem[i0] == x )
      return i0;
    if( this->mem[i1] == x )
      return i1;
  }
  return not_an_index;
```

```
  }
};

#endif
```

A.7 cuckoo-dary.h

In dieser Datei ist das *d*-äre Cuckoo Hashing (*cuckoo-dary*) definiert.

```
#ifndef __cuckoo_dary_h__
#define __cuckoo_dary_h__

#include "cuckoo-common.h"

template <class K, class HF, class EMPTY>
class CuckooDary : public CuckooDictionary<K,HF,EMPTY>{
protected:
  HF *h;
  int d1;

public:
  CuckooDary( int _d = 4, const double &_epsilon=0.1 ):
  CuckooDictionary<K,HF,EMPTY>( _d, _epsilon ){
    d1 = d-1;
    h = new HF[_d];
    reserve( 4*this->d );
  };

  ~CuckooDary( ) {
    delete [] h;
  };

  int insert( const K &_x ) {
    K x( _x );
    int freecell = not_an_index;
    int i = 0;

    int r = my_rand( ) % this->d;
    for( int j = this->d-1; j >= 0; j-- ) {
      i = h[r]( x );
      if( this->mem[i] == x )
        return found;
      else
        if( freecell == not_an_index &&
            this->__empty.is_empty( this->mem[i] ) )
          freecell = i;
      r++;
      if( r == this->d )
        r -= this->d;
    }

    this->account -= this->d;

    if( freecell != not_an_index ) {
      this->mem[freecell] = x;
      this->m_nkeys++;
      return freecell;
    }

    while( --this->account > 0 ) {
      ::swap( x, this->mem[i] );
      if( this->__empty.is_empty(x) ) {
        this->m_nkeys++;
        return i;
      }
      r = my_rand( ) % d1;
      int oldi = i;
      i = h[r]( x );
      if( i == oldi )
        i = h[d1]( x );
    }
    this->remaining_key = x;
    return not_an_index;
  }

  void remove
    ( const K & _x ) {
    int index = lookup( _x );
    if( index != not_an_index ) {
      this->mem[index] = this->__empty;
      this->m_nkeys--;
    }
  }

  int insert_grow( const K & _x ) {
    this->account = this->insert_grow_account;
    int index = insert( _x );
    if( index != not_an_index )
      return index;
    if( !grow_by( this->f_grow ) )
      return not_an_index;
    return insert( this->remaining_key );
  };

  /** Versucht, ncells Zellen zu reservieren
      und die Schlüssel dort unterzubringen.
      Wenn das nicht geht, wird false zurückgegeben. */
  bool reserve( int ncells ) {
    HF *_h = h;
    h = new HF[this->d];
    int oldm = this->m;
    int oldnkeys = this->m_nkeys;
    K old_remaining_key = this->remaining_key;
    int old_account = this->account;
    for( int j = this->d-1; j >= 0; j-- ) {
      h[ j ] = HF( ncells / this->d, j * (ncells/this->d) );
    }
    K *oldmem = this->mem;
    this->mem = new K[ ncells ];
    this->m = ncells;
    this->set_account( (int)(this->account_factor * this->m) );
    this->clear( );
    this->remaining_key = old_remaining_key;
    if( oldmem != NULL ) {
      for( int j = oldm-1; j >= 0; j-- ) {
        if( !this->__empty.is_empty(oldmem[j]) ) {
          if( insert(oldmem[j]) == not_an_index ) {
            delete[ ] h;
            delete[ ] this->mem;
            h = _h;
            this->mem = oldmem;
            this->m = oldm;
            this->m_nkeys = oldnkeys;
            this->remaining_key = old_remaining_key;
```

```
            this->account = old_account;
            return false;
          }
        }
      }
    }
    delete[ ] oldmem;
  }
  delete[ ] _h;
  return true;
};

bool grow_by( const double &factor ) {
  int _m = (int)ceil( factor * this->m );
  if( factor > 1 && _m == this->m )
    _m++;
  return reserve( _m );
}

inline int lookup( const K &x ) const {
  for( int j = this->d-1; j >= 0; j-- ) {
    int i = h[j]( x );
    if( this->mem[i] == x )
      return i;
  }
  return not_an_index;
}

double get_avg_probes( ) const {
  return max( 2*this->d+1., 2/log(1.+this->epsilon));
};

double get_element_probes( ) const {
  double res =
    2*std::max(
      this->d+1.,
      ::pow(1/this->epsilon, ::log(2.0*this->d)) );
  return res;
};
};

#endif
```

A.8 lp.h

In dieser Datei ist das Lineare Sondieren (*lp*) definiert.

```
#ifndef __lp_h__
#define __lp_h__

#include "dictionary-common.h"

template <class K, class HF, class EMPTY>
class LP : public Dictionary<K, HF, EMPTY> {
protected:
  HF h;
  int grow_at;

public:
  LP( const double &_epsilon=0.1 ) :
    Dictionary<K,HF,EMPTY>( _epsilon ) {
    reserve( 16 );
};

int insert( const K &_x ) {
  if( this->m_nkeys >= this->m ) {
    this->remaining_key = _x;
    return not_an_index;
  }
  int i = h(_x);
  /* Wir suchen das erste freie oder gelöschte Feld
  oder brechen ab, falls _x gefunden wird */
  while( !this->__empty.is_empty(this->mem[i])
         && !(this->mem[i] == _x) ) {
    i--;
    if( i < 0 )
      i = this->m-1;
  }
  if( this->mem[i] == _x )
    return found;
  this->mem[i] = _x;
  this->m_nkeys++;
  return i;
}

void remove( const K & _x ) {
  int i = lookup( _x );
  if( i == not_an_index )
    return;
  this->mem[ i ] = this->__empty;
  int j = i;
  i--;
  if( i < 0 )
    i = this->m-1;
  while( !this->__empty.is_empty(this->mem[i]) ) {
    int r = h( this->mem[i] );
    if( ( r >= j || j >= i ) && ( j >= i || i > r )
        && ( i > r || r >= j ) ) {
      this->mem[j] = this->mem[i];
      this->mem[i] = this->__empty;
      j = i;
    }
    i--;
    if( i < 0 )
      i = this->m-1;
  }
  this->m_nkeys--;
}

int insert_grow( const K & _x ) {
  if( this->remaining_nkeys >= grow_at ) {
    grow_by( this->f_grow );
  }
  return insert( _x );
};

void clear( ) {
  for( int j = this->m-1; j >= 0; j-- )
    this->mem[ j ] = this->__empty;
  this->m_nkeys = 0;
  this->remaining_key = this->__empty;
}

/** Versucht, ncells Zellen zu reservieren
und die Schlüssel dort unterzubringen.
```

```
    Wenn das nicht geht, wird false zurückgegeben. */
80  bool reserve( int ncells ) {
      HF _h = h;
      int oldm = this->m;
      int oldnkeys = this->m_nkeys;
      K old_remaining_key = this->remaining_key;
85
      h = HF( ncells );
      K *oldmem = this->mem;
      this->mem = new K[ ncells ];
      this->m = ncells;
90    this->clear( );
      this->remaining_key = old_remaining_key;
      if( oldmem != NULL ) {
        for( int j = oldm-1; j >= 0; j-- ) {
          if( oldmem[j] >= 0 ) {
95          if( insert(oldmem[j]) == not_an_index ){
              h = _h;
              this->m = oldm;
              this->m_nkeys = oldnkeys;
              this->remaining_key = old_remaining_key;
100           delete[ ] this->mem;
              this->mem = oldmem;
              return false;
            }
          }
105     }
        delete[ ] oldmem;
      }
      return true;
    };
110
    bool grow_by( const double &factor ) {
      int _m = (int)ceil( factor * this->m );
      if( factor > 1 && _m == this->m )
        _m++;
115   return reserve( _m );
    }

    inline int lookup( const K &_x ) const {
      int i = h(_x);
120   while( !this->__empty.is_empty(this->mem[i]) ){
        if( this->mem[i] == _x )
          return i;
        i--;
        if( i < 0 )
125       i = this->m-1;
      }
      return not_an_index;
    }
  };
130
  #endif
```

A.9 main.cpp

In dieser Datei werden die einzelnen Verfahren getestet.

```
  #include "cuckoo-dary.h"
  #include "cuckoo-block.h"
  #include "cuckoo-block-simple.h"
5 #include "lp.h"

  #include "keygen.h"

  #include <sys/times.h>
10 #include <sys/time.h>
  #include <sys/resource.h>
  #include <unistd.h>
  #include <iostream>
  #include <algorithm>
15 #include <math.h>

  using namespace std;

  int nkeys = 10000;
20 double epsilon = 1;
  int verfahren = 0;
  int d = 8;

  vector<int> keys;
25
  tms tms_start, tms_stop;

  inline void start( ) {
    times( &tms_start );
30 };

  inline clock_t stop( ) {
    times( &tms_stop );
    return tms_stop.tms_stime + tms_stop.tms_utime
35         - tms_start.tms_stime - tms_start.tms_utime;
  };

  template < class DICT >
  void test_insert( DICT dict ) {
40  clock_t t;

    double account_faktor = dict.get_avg_probes( );
    int the_account = (int)(account_faktor*nkeys);
    dict.reserve( (int) ((1+epsilon) * nkeys) );
45  dict.set_account( the_account );
    clog << "Einfügen bei Platz " << 1+epsilon
    << " [account = " << the_account << "]" << endl;
    start();
    int index = 0;
50  for( int i = keys.size( )-1; i >= 0; i-- ) {
      index = dict.insert( keys[i] );
      if( index == not_an_index )
        break;
    }
55  t = stop( );
    if( index == not_an_index )
      cout << "Zeit (alle) [sek]: ***" << endl;
    else
      cout << "Zeit (alle) [sek]: "
60    << double(t) / sysconf( _SC_CLK_TCK ) << endl;
    clog << "Restkonto: " << dict.get_account( ) << endl;

    clog << "Loadfactor: " << dict.get_loadfactor( ) << endl;
    clog << "Positive Lookups: ";
65  start();
    for( int i = keys.size( )-1; i >= 0; i-- ) {
```

```
    if( dict.lookup( keys[i] ) == -1 ) {
      cerr << "Fehler beim Nachschlagen!" << endl;
      clog << flush;
      exit( -1 );
    }
  }
  t = stop( );
  cout << "Pos. Lookups [sek]: " <<
  double(t) / sysconf( _SC_CLK_TCK ) << endl;

  clog << "Negative Lookups: ";
  /* Wir testen hier nkeys Schlüssel,
  die garantiert nicht im Wörterbuch sind.*/
  int neg = *(keys.end( )-1)+1;
  start();
  for( int i = keys.size( )-1; i >= 0; i--, neg++ ) {
    if( dict.lookup( neg ) != -1 ) {
      cerr << "Fehler beim Nachschlagen!" << endl;
      clog << flush;
      exit( -1 );
    }
  }
  t = stop( );
  cout << "Neg. Lookups [sek]: "
  << double(t) / sysconf( _SC_CLK_TCK ) << endl;

  account_faktor = dict.get_element_probes( );
  the_account = (int)(account_faktor*log((double)nkeys));
  clog << "Löschen/Einfügen: "
  << "[account = " << the_account << "]" << endl;
  neg = *(keys.end( )-1)+1;
  start();
  for( int i = keys.size( )-1; i >= 0; i--, neg++ ) {
    dict.remove( keys[i] );
    dict.set_account( the_account );
    index = dict.insert( neg );
    if( index == not_an_index )
      break;
  }
  t = stop( );
  if( index == not_an_index )
    cout << "Zeit (remove/insert) [sek]: ***" << endl;
  else
    cout << "Zeit (remove/insert) [sek]: "
    << double(t) / sysconf( _SC_CLK_TCK ) << endl;
  clog << "Größe des Wörterbuches: "
  << dict.get_nkeys( ) << endl;

  clog << "Lookups nach remove/insert: ";
  neg = *(keys.end( )-1)+1;
  start();
  for( int i = keys.size( )-1; i >= 0; i--, neg++ ) {
    if( ( dict.lookup( neg ) == not_an_index ) ||
        ( dict.lookup( keys[i] ) != not_an_index ) ) {
      cerr << "Fehler beim Nachschlagen!" << endl;
      clog << flush;
      exit( -1 );
    }
  }
  t = stop( );
  cout << "Lookups nach remove [sek]: "
  << double(t) / sysconf( _SC_CLK_TCK ) << endl;
}


const int HF_DEG = 3;

#define CUCKOO_LP 1
#define CUCKOO_DARY 2
#define CUCKOO_BLOCK 3
#define CUCKOO_BLOCK_SIMPLE 4
#define DICT_LP 5


typedef intStaticHashFun_poly_offset<HF_DEG> HF_offset;
typedef intStaticHashFun_poly<HF_DEG> HF;

int main(int argc, char *argv[]) {
  if( argc < 2 ) {
    exit( -1 );
  }
  time_t t;
  srand48( time( &t ) );
  if( argc >= 2 )
    verfahren = atoi( argv[ 1 ] );
  if( argc >= 3 )
    nkeys = atoi( argv[ 2 ] );
  if( argc >= 4 )
    epsilon = atof( argv[ 3 ] );
  if( argc >= 5 )
    d = atoi( argv[ 4 ] );

  keys = KeyGen<int>(nkeys)();
  switch( verfahren ) {

  case CUCKOO_DARY :
    test_insert( CuckooDary<int, HF_offset,
                 int_empty_cell> ( d, epsilon ) );
    break;
  case CUCKOO_BLOCK :
    test_insert( CuckooBlock<int, HF,
                 int_empty_cell> ( d, epsilon ) );
    break;
  case CUCKOO_BLOCK_SIMPLE :
    test_insert( CuckooBlockSimple<int,HashFunDouble<int,HF>,
                 int_empty_cell> ( d, epsilon ) );
    break;
  case CUCKOO_LP :
    test_insert( CuckooLP<int, HF,
                 int_empty_cell>( d, epsilon ) );
    break;
  case DICT_LP :
    test_insert( LP<int, HF,
                 int_empty_cell> ( epsilon ) );
    break;
  }
  return 0;
}
```

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis