Randomisierte Algorithmen

Thomas Worsch
Fakultät für Informatik
Karlsruher Institut für Karlsruhe

Vorlesungsunterlagen
Wintersemester 2015/2016

Vorläufiges Inhaltsverzeichnis

1 Einleitung 4
- 1.1 Grundsätzliches 4
- 1.2 Einige Schlagwörter 5
- 1.3 Geschichtliches 6
- 1.4 Einige Literaturhinweise 7

2 Erste Beispiele 10
- 2.1 Ein randomisierter Identitätstest 10
- 2.2 Vergleich von Wörtern 12
- 2.3 Ein randomisierter Quicksortalgorithmus 15

3 Probabilistische Komplexitätsklassen 18
- 3.1 Probabilistische Turingmaschinen 18
- 3.2 Komplexitätsklassen 19
- 3.3 Beziehungen zwischen Komplexitätsklassen 23

4 Routing in Hyperwürfeln 26
- 4.1 Das Problem und ein deterministischer Algorithmus 26
- 4.2 Markov- und Chebyshev-Ungleichung 29
- 4.3 Chernoff-Schranken 29
- 4.4 Erster randomisierter Algorithmus 33
- 4.5 Die probabilistische Methode 37
- 4.6 Zweiter randomisierter Algorithmus 37

5 Zwei spieltheoretische Aspekte 41
- 5.1 Und-Oder-Bäume und ihre deterministische Auswertung 41
- 5.2 Analyse eines randomisierten Algorithmus für die Auswertung von UOB 42
- 5.3 Zwei-Personen-Nullsummen-Spiele 44
- 5.4 Untere Schranken für randomisierte Algorithmen 45

6 Graph-Algorithmen 48
- 6.1 Minimale Schnitte 48
- 6.2 Minimale spannende Bäume 55
  - 6.2.1 Ein deterministischer Algorithmus für MST 55
  - 6.2.2 F-leichte und F-schwere Kanten 57
  - 6.2.3 Ein randomisierter MSF-Algorithmus 57

7 Random Walks 61
- 7.1 Ein randomisierter Algorithmus für 2-SAT 61
- 7.2 Random Walks 62
- 7.3 Widerstandsnetzwerke 63
- 7.4 Randomisierte Algorithmen für Zusammenhangstests 66

8 Markov-Ketten **69**
8.1 Grundlegendes zu Markov-Ketten 69
8.2 Irreduzible und ergodische Markov-Ketten 71

9 Schnell mischende Markov-Ketten **76**
9.1 Der Metropolis-Algorithmus 76
9.2 Anmerkungen zu Eigenwerten 78
9.3 Schnell mischende Markov-Ketten 78

10 Randomisiertes Approximieren **84**
10.1 Polynomielle Approximationsschemata 84
10.2 Zählen von Lösungen von Formeln in DNF 85

11 Online-Algorithmen am Beispiel des Seitenwechselproblems **90**
11.1 Das Seitenwechselproblem und deterministische Algorithmen 90
11.2 Randomisierte Online-Algorithmen und Widersacher 92
11.3 Seitenwechsel gegen einen unwissenden Widersacher 93
11.4 Einschub: Amortisierte Analyse 97
11.5 Seitenwechsel gegen adaptive Widersacher 98

12 Hashing **105**
12.1 Grundlagen 105
12.2 Universelle Familien von Hashfunktionen 105
12.3 Zwei 2-universelle Familien von Hashfunktionen 107
12.4 Das dynamische Wörterbuchproblem 109
12.5 Das statische Wörterbuchproblem 110
12.6 Derandomisierung 112

13 Pseudo-Zufallszahlen **117**
13.1 Allgemeines 117
13.2 Generatoren für Pseudo-Zufallszahlen 119
13.2.1 Middle-square- und Muddle-square-Generator 119
13.2.2 Lineare Kongruenzgeneratoren 119
13.2.3 Mehrfach rekursive Generatoren 120
13.2.4 Inverse Kongruenzgeneratoren 120
13.2.5 Mersenne-Twister 121
13.2.6 Zellularautomaten als Generatoren 121
13.2.7 Kombination mehrerer Generatoren 122
13.3 Tests für Pseudo-Zufallszahlen 123
13.3.1 Grundsätzliches: χ^2- und KS-Test 123
13.3.2 Einfache empirische Tests 124
13.3.3 Weitere Tests 126

A Wahrscheinlichkeitstheoretische Grundlagen **129**
A.1 Allgemeines 129
A.2 Zufallsvariablen 130

 © Th. Worsch 2000-2015

1 Einleitung

1.1 Grundsätzliches

1.1 *Randomisierte* (oder auch *probabilistische*) *Algorithmen* sind Rechenverfahren, bei denen man den klassischen Algorithmenbegriff aufweicht.

Eine mögliche Sichtweise ist die, dass als ein möglicher algorithmischer Elementarschritt ein *zufälliger Wert* „irgendwie" zur Verfügung gestellt wird.

Eine andere Sichtweise ist die, dass nicht mehr verlangt wird, dass stets *ein*deutig festgelegt ist, welches der als nächstes auszuführende Elementarschritt ist. Im Unterschied zu nichtdeterministischen Verfahren ist aber quantifiziert, welche Fortsetzung mit welcher Wahrscheinlichkeit gewählt wird.

Eine dritte Sichtweise ist die eines deterministischen Algorithmus, der aber neben der „eigentlichen" Eingabe auch noch als „uneigentliche" Eingabe einen Zufallswert (etwa eine Reihe von Zufallsbits) bekommt. Auf diese Weise wird sozusagen aus einer Vielzahl von deterministischen Algorithmen jeweils einer ausgewählt.

1.2 Allen diesen Sichtweisen gemeinsam ist die Tatsache, dass sich bei mehreren Ausführungen des randomisierten Algorithmus *für die gleiche Eingabe*[1] *verschiedene Berechnungen* ergeben können.

1.3 Hieraus folgt zum Beispiel, dass im allgemeinen selbst für eine einzelne festgehaltene Eingabe im allgemeinen *das berechnete Ergebnis, die Laufzeit und der Speicherplatzbedarf nicht präzise anzugeben* sind. Vielmehr handelt es sich dabei um *Zufallsvariablen*.

Man wird dann etwa daran interessiert sein, die Erwartungswerte dieser Größen herauszufinden. Unter Umständen sind auch weitergehende Aussagen interessant, zum Beispiel die, dass (in einem später noch zu definierenden mathematisch präzisen Sinne) „mit hoher Wahrscheinlichkeit" Abweichungen vom Erwartungswert „sehr klein" sind. Die Erwartungswerte etc. möchte man wie üblich in Abhängigkeit nur von der Größe der Eingabe bestimmen.

1.4 Man geht sogar so weit, von einem randomisierten Algorithmus für ein Problem zu sprechen, wenn die Ausgaben manchmal, aber eben „hinreichend selten" im Sinne der Problemspezifikation *falsch* sind.

Je nach bisher genossener Informatikausbildung mag dies im ersten Moment äußerst befremdlich erscheinen. Ohne jeden Zweifel kommt hier ein (im Vergleich zu deterministischen Algorithmen) neuer Standpunkt ins Spiel. Dass der aber nicht völlig aus der Luft gegriffen ist, mag durch die folgende Überlegung verdeutlicht werden (Niedermeier 1997):

Nimmt man an, dass jedes Jahrtausend mindestens einmal durch einen Meteoriteneinschlag auf der Erde eine Fläche von $100\ \mathrm{m}^2$ verwüstet wird, dann wird ein Rechner während einer Mikrosekunde seiner Arbeit mit einer Wahrscheinlichkeit von mindestens 2^{-100} zerstört. Andererseits gibt es randomisierte Algorithmen, die auf einem Rechner in Sekundenschnelle die

[1] Damit sind hier natürlich „eigentliche" Eingaben gemeint.

Ausgabe produzieren: „Die größte Primzahl kleiner als 2^{400} ist $2^{400} - 593$.“. Und wenn diese Aussage geliefert wird, ist sie nur mit Wahrscheinlichkeit 2^{-100} falsch.

1.5 Unter Umständen etwas weniger irritierend ist die Situation bei *Optimierungsalgorithmen*. Hier ist es häufig so, dass die Berechnung von Lösungen, die in einem gewissen Sinne optimal sind, (zumindest mit allen bekannten Verfahren) zu lange dauern würde, und man daher mit Algorithmen zufrieden ist, die zwar suboptimale aber immerhin noch akzeptable Lösungen liefern.

Auch hierzu können randomisierte Algorithmen eingesetzt werden. Eine Frage, die dann vermutlich eine Rolle spielt, ist, ob man eventuell zeigen kann, dass Lösungen einer gewissen Qualität mit einer gewissen Wahrscheinlichkeit gefunden werden.

1.6 Weshalb ist man an randomisierten Algorithmen interessiert? Manchmal gibt es randomisierte Algorithmen zur Lösung eines Problemes, die genauso gut sind wie die besten deterministischen, aber weitaus leichter zu formulieren und implementieren. Manchmal kennt man randomisierte Algorithmen, die sogar „besser“ sind als die besten deterministischen Algorithmen. In einigen Fällen ist es sogar so, dass es für ein Problem zwar keinen deterministischen Algorithmus gibt, der es löst, aber einen randomisierten, der (in gewissem Sinne) das Gewünschte leistet.

Es sei aber noch einmal erwähnt, dass man dafür häufig einen Preis bezahlt, dass nämlich Ausgaben falsch sein können. Mit anderen Worten wird der Begriff der Lösung eines Problems unter Umständen nicht mehr mit der gleichen Strenge wie bei deterministischen Algorithmen benutzt. Es gibt aber auch randomisierte Algorithmen, die immer das richtige Ergebnis liefern.

1.7 Man beachte, dass bei randomisierten Algorithmen zum Beispiel eine Aussage über den Erwartungswert der Laufzeit *für alle Eingaben* korrekt zu sein hat.

Dies ist im Unterschied etwa zur *probabilistischen Analyse deterministischer Algorithmen* zu sehen. Dort wird eine gewisse Wahrscheinlichkeitsverteilung für die Eingaben zu Grunde gelegt, und zum Beispiel daraus und aus den Laufzeiten für die einzelnen Eingaben ein (anderer!) Erwartungswert für die Laufzeiten ermittelt („average-case complexity“).

1.2 Einige Schlagwörter

Wie kann „Zufall“ vorteilhaft in einem randomisierten Algorithmus eingesetzt werden? Im Laufe der Zeit hat sich Reihe von „Standardmethoden“ gefunden, um Zufallsbits anzuwenden. Einige der in der folgenden Liste aufgeführten werden im Laufe der Vorlesung noch in Beispielalgorithmen auftreten und analysiert werden. Andere seien hier der Vollständigkeit halber aus der Arbeit von Karp (1991) mit aufgeführt:

Fingerabdrücke: Hierunter hat man sich kompakte „Repräsentationen“ großer Daten vorzustellen. Durch Benutzung einer zufälligen Komponente bei der Berechnung kann man erreichen, dass (in einem gewissen Rahmen) identische Fingerabdrücke tatsächlich auf identische Ausgangsdaten hinweisen.

Zufälliges Auswählen und Umordnen: Durch die Benutzung zufällig ausgewählter Teilmengen, zufälliger Umordnungen, o.ä. kann man mitunter mit großer Wahrscheinlichkeit Eingaben, die im deterministischen Fall zu schlechten Ausreißern z. B. bei der Laufzeit führen, „unschädlich machen“.

 © Th. Worsch 2000-2015

Überfluß an Zeugen: Mitunter gibt es für eine gewisse Eigenschaft von Eingabedaten „Zeugen“. Mit einem deterministischen Algorithmus einen von ihnen zu finden ist unter Umständen sehr schwer, obwohl es „sehr viele“ gibt. Deshalb kann die zufällige Wahl irgendeines Wertes schnell einen Zeugen liefern. Wenn dann auch noch die Zeugeneigenschaft leicht nachzuweisen ist . . .

Vereitelung gegnerischer Angriffe: Im Zusammenhang mit der Komplexität eines Problems kann man den Wert eines Zwei-Personen-Nullsummen-Spieles des Algorithmenentwerfers gegen einen „Bösewicht“ betrachten, der versucht, durch die Auswahl „schlechter“ Eingaben ineffiziente Berechnungen des Algorithmus zu erzwingen. Durch zufällige Entscheidungen zwischen verschiedenen Algorithmen (siehe die dritte Sichtweise in Punkt 1.1) hat es der Bösewicht unter Umständen schwerer.

Lastbalancierung: In parallelen Algorithmen kann randomisierte Lastverteilung gute Dienste leisten.

Schnell mischende Markovketten: Bei der Konstruktion eines „zufälligen Objektes“ muß manchmal sichergestellt werden, dass man schnell genug eines mit den gewünschten Eigenschaften findet. Hierzu bedient man sich der Theorie der Markovketten und zeigt, dass sie die dem Algorithmus zu Grunde liegende „schöne Eigenschaften“ hat und schnell gegen die stationäre Verteilung konvergiert.

Isolierung und Symmetriebrechung: Wenn in einer verteilten Umgebung alle nach dem gleichen Algorithmus vorgehen und nicht durch eindeutige Namen bzw. Nummern unterschieden sind, sich aber trotzdem Unterschiede ergeben sollen, läßt sich dies z. B. durch Randomisierung erreichen.

Weitere Aspekte, die im Zusammenhang mit randomisierten Algorithmen eine Rolle spielen und auf die wir zum Teil auch eingehen werden, sind unter anderem:

probabilistische Methode: Das ist ein Werkzeug, um die Existenz gewisser Objekte nachzuweisen, indem man zeigt, dass die Wahrscheinlichkeit, bei einem bestimmten randomisierten Algorithmus auf ein Objekt der gewünschten Art zu stoßen, echt größer Null ist.

randomisierte Komplexitätsklassen: So wie im Nicht-/Deterministischen Komplexitätsklassen wie **P** oder **NP** eine wichtige Rolle spielen, haben sich für komplexitätstheoretische Untersuchungen randomisierter Algorithmen unter anderem die Modelle probabilistischer Turingmaschinen und Schaltkreise bewährt und mit **BPP**, **ZPP**, **RP** oder **RNC** bezeichnete Komplexitätsklassen Bedeutung erlangt.

Zufall als Berechnungsressource: Woher bekommt man Zufallsbits? Sind Zufallsbits eigentlich „umsonst“?

Elimination von Zufall: Falls nicht, gibt es Möglichkeiten, ihren Gebrauch einzuschränken, ohne algorithmische Nachteile in Kauf nehmen zu müssen?

1.3 Geschichtliches

Wie so häufig gibt es nicht genau einen exakt bestimmbaren Punkt, an dem zum ersten Mal das Konzept eines randomisierten Algorithmus eingeführt und verwendet wurde.

 © Th. Worsch 2000-2015

Shallit 1992 hat die Informatik darauf hingewiesen, dass in wenigstens zwei Kulturen, die als „primitiv" zu bezeichnen man sich immer noch gelegentlich herablässt, in Verfahren zur Bestimmung eines bestimmten zukünftigen Verhaltens (z. B. „Wo soll gejagt werden?") Ansätze vorhanden sind, denen zumindest eine Verwandtschaft zu randomisierten Algorithmen und den dort angewendeten Methoden nicht abgesprochen werden kann.

Als erstes Beispiel eines randomisierten Algorithmus betrachten wir im folgenden ein Verfahren des zentralafrikanischen Stammes der Azande. Sie befragen ein „Orakel", indem einem Huhn eine giftige rote Paste verabreicht wird. Üblicherweise wird das Huhn dann eine Zeit lang an heftigen Krämpfen leiden. Manchmal wird es sogar sterben, manchmal aber auch überhaupt keine Reaktion zeigen.

Die Tatsache, dass ein Huhn überlebt bzw. nicht, wird als Antwort des Orakels auf eine vorher gestellte Frage interpretiert. Man geht davon aus, dass das Orakel nie irrt, sofern man es korrekt befragt, d. h. die richtige Menge des Giftes verabreicht hat. Es kann nicht ausgeschlossen werden, dass einmal „aus Versehen" eine zu kleine oder zu große Menge Giftes benutzt wurde. Jede Orakelbefragung beinhaltet also sozusagen ein zufälliges Element. Um damit fertig zu werden, wenden die Azande folgendes Verfahren an:

> Sie formulieren die Anfrage zweimal. Einmal so, dass die Bestätigung durch das Orakel durch den Tod eines Huhns zum Ausdruck käme, und einmal so, dass die Bestätigung durch das Orakel durch das Überleben eines Huhns zum Ausdruck käme. Nur wenn bei den beiden Orakelbefragungen in genau einem Fall ein Huhn stirbt, wird die Antwort als gültig akzeptiert. Sterben beide oder kein Huhn, wird die Antwort des Orakels verworfen.

Da die Hühner eine wertvolle Ressource für den Stamm darstellen, wendet er darüberhinaus Methoden an, um damit sparsam umzugehen. So wird z. B. auf die zweite Befragung verzichtet, wenn das Huhn bei der ersten sehr schnell stirbt. Manchmal werden zunächst die ersten Orakelantworten für zwei verschiedene Fragen ermittelt. Falls bei beiden ein Huhn stirbt, wird nur ein (drittes) Huhn benutzt, um beide bestätigen zu lassen. Schließlich merkt O. K. Moore 1957 in einer Fußnote einer Arbeit zu diesem Thema an: „Incidentally, their manner of framing questions—they use complex conditionals—so as to obtain as many definitive answers while sacrificing as few fowls as possible, would do credit to a logician." Was damit genau gemeint ist, ist unklar. Shallit (1992) mutmaßt, dabei könne es sich um so etwas wie binäre Suche handeln, indem z. B. drei Hühner geopfert werden, um eine von acht Möglichkeiten auszuwählen.

Die anscheinend erste Arbeit über probabilistische Maschinen stammt von Leeuw u. a. 1955. Weitere grundlegende automatentheoretische Arbeiten sind die von Rabin (1963) über probabilistische endliche Automaten und die von Gill (1977) über probabilistische Turingmaschinen. Auf dieses Modell werden wir im Zusammenhang mit komplexitätstheoretischen Betrachtungen zurückkommen.

Einige der ersten randomisierten Algorithmen überprüfen (in einem gewissen Sinne) zahlentheoretische Eigenschaften. Solovay und Strassen (1977), Solovay und Strassen (1978) und Rabin (1976) haben Verfahren angegeben, die eine eingegebene Zahl auf Zusammengesetztheit bzw. Primheit überprüfen. Wir werden später darauf zurückkommen.

1.4 Einige Literaturhinweise

Das Standardbuch über randomisierte Algorithmen ist

 © Th. Worsch 2000-2015

- Rajeev Motwani und Prabhakar Raghavan (1995). *Randomized Algorithms*. Cambridge University Press. ISBN: 0-521-47465-5.

Außerdem sind die folgenden Bücher zumindest für einen Teil der Vorlesung relevant und empfehlenswert:

- Allan Borodin und Ran El-Yaniv (1998). *Online Computation and Competitive Analysis*. Cambridge Unviersity Press. ISBN: 0-521-56392-5.
- Juraj Hromkovič (2004). *Randomisierte Algorithmen: Methoden zum Entwurf von zufallsgesteuerten Systemen für Einsteiger*. Teubner-Verlag. ISBN: 3-519-00470-4.

Abschnitte bzw. Kapitel über komplexitätstheoretische Aspekte finden sich zum Beispiel auch in den folgenden Büchern:

- Giorgio Ausiello u. a. (1999). *Complexity and Approximation*. Berlin: Springer. ISBN: 3-540-65431-3.
- Daniel Pierre Bovet und Pierluigi Crescenzi (1994). *Introduction to the Theory of Complexity*. New York: Prentice Hall.
- Jozef Gruska (1997). *Foundations of Computing*. International Thomson Computer Press.
- Christos H. Papadimitriou (1994). *Computational Complexity*. Addison-Wesley.

Um erste Eindrücke von dem Gebiet zu bekommen, sind die folgenden Übersichtsartikel hilfreich:

- Richard M. Karp (1991). „An introduction to randomized algorithms“. In: *Discrete Applied Mathematics* 34, S. 165–201.
- Rajiv Gupta, Scott A. Smolka und Shaji Bhaskar (1994). „On Randomization in Sequential and Distributed Algorithms“. In: *ACM Computing Surveys* 26.1, S. 7–86.
- Rajeev Motwani und Prabhakar Raghavan (1997). „Randomized Algorithms“. In: *The Computer Science and Engineering Handbook*. Hrsg. von A. B. Tucker Jr. CRC Press. Kap. 7, S. 141–161.

Schließlich finden sich im WWW eine Reihe von Vorlesungsskripten über randomisierte Algorithmen. Zum Beispiel:

- Josep Diaz (1997). *Probabilistic Algorithms*. Im WWW zugänglich unter `http://www.lsi.upc.es/~diaz/ralcom.ps.gz`.
- Michel X. Goemans (1994). *Randomized Algorithms*. Im WWW zugänglich unter `ftp://ftp.theory.lcs.mit.edu/pub/classes/18.415/notes-random.ps`.
- Seffi Naor (1993). *Probabilistic Methods in Computer Science*. Im WWW zugänglich unter `http://www.uni-paderborn.de/fachbereich/AG/agmadh/Scripts/GENERAL/naor.notes.ps.gz`.
- Rolf Niedermeier (1997). *Randomisierte Algorithmen*. Im WWW zugänglich unter `http://www-fs.informatik.uni-tuebingen.de/~niedermr/teaching/ra-script.ps.Z`.

Einige dieser Skripte sind auch über `http://www.uni-paderborn.de/fachbereich/AG/agmadh/WWW/scripts.html` zugänglich und als Kopie von dort schneller zu erhalten.

 © Th. Worsch 2000-2015

Literatur

Ausiello, Giorgio u. a. (1999). *Complexity and Approximation*. Berlin: Springer. ISBN: 3-540-65431-3 (siehe S. 8).

Borodin, Allan und Ran El-Yaniv (1998). *Online Computation and Competitive Analysis*. Cambridge Unviersity Press. ISBN: 0-521-56392-5 (siehe S. 8, 90).

Bovet, Daniel Pierre und Pierluigi Crescenzi (1994). *Introduction to the Theory of Complexity*. New York: Prentice Hall (siehe S. 8).

Diaz, Josep (1997). *Probabilistic Algorithms*. Im WWW zugänglich unter `http://www.lsi.upc.es/~diaz/ralcom.ps.gz` (siehe S. 8).

Gill, John (1977). „Computational complexity of probabilistic Turing machines". In: *SIAM Journal on Computing* 6.4, S. 675–695 (siehe S. 7).

Goemans, Michel X. (1994). *Randomized Algorithms*. Im WWW zugänglich unter `ftp://ftp.theory.lcs.mit.edu/pub/classes/18.415/notes-random.ps` (siehe S. 8).

Gruska, Jozef (1997). *Foundations of Computing*. International Thomson Computer Press (siehe S. 8).

Gupta, Rajiv, Scott A. Smolka und Shaji Bhaskar (1994). „On Randomization in Sequential and Distributed Algorithms". In: *ACM Computing Surveys* 26.1, S. 7–86 (siehe S. 8).

Hromkovič, Juraj (2004). *Randomisierte Algorithmen: Methoden zum Entwurf von zufallsgesteuerten Systemen für Einsteiger*. Teubner-Verlag. ISBN: 3-519-00470-4 (siehe S. 8).

Karp, Richard M. (1991). „An introduction to randomized algorithms". In: *Discrete Applied Mathematics* 34, S. 165–201 (siehe S. 5, 8).

Leeuw, K. de u. a. (1955). „Computability by probabilistic machines". In: *Automata Studies*. Hrsg. von C. E. Shannon und J. McCarthy. Princeton Unviersity Press, S. 183–212 (siehe S. 7).

Moore, Omar Khayyam (1957). „Divination—a new perspective". In: *American Anthropologist* 59, S. 69–74 (siehe S. 7).

Motwani, Rajeev und Prabhakar Raghavan (1995). *Randomized Algorithms*. Cambridge University Press. ISBN: 0-521-47465-5 (siehe S. 8, 96, 102).

— (1997). „Randomized Algorithms". In: *The Computer Science and Engineering Handbook*. Hrsg. von A. B. Tucker Jr. CRC Press. Kap. 7, S. 141–161 (siehe S. 8).

Naor, Seffi (1993). *Probabilistic Methods in Computer Science*. Im WWW zugänglich unter `http://www.uni-paderborn.de/fachbereich/AG/agmadh/Scripts/GENERAL/naor.notes.ps.gz` (siehe S. 8).

Niedermeier, Rolf (1997). *Randomisierte Algorithmen*. Im WWW zugänglich unter `http://www-fs.informatik.uni-tuebingen.de/~niedermr/teaching/ra-script.ps.Z` (siehe S. 4, 8).

Papadimitriou, Christos H. (1994). *Computational Complexity*. Addison-Wesley (siehe S. 8).

Rabin, Michael O. (1963). „Probabilistic Automata". In: *Information and Control* 6, S. 230–245 (siehe S. 7).

— (1976). „Probabilistic Algorithms". In: *Algorithms and Complexity*. Hrsg. von J. F. Traub. Academic Press, S. 21–39 (siehe S. 7, 17, 20).

Shallit, Jeffrey (1992). „Randomized Algorithms in "Primitive" Cultures or What is the Oracle Complexity of a Dead Chicken". In: *ACM SIGACT News* 23.4, S. 77–80 (siehe S. 7).

Solovay, R. und V. Strassen (1977). „A Fast Monte-Carlo Test for Primality". In: *SIAM Journal on Computing* 6.1, S. 84–85 (siehe S. 7, 17).

 © Th. Worsch 2000-2015

— (1978). „Erratum: A Fast Monte-Carlo Test for Primality“. In: *SIAM Journal on Computing* 7.1, S. 118 (siehe S. 7, 17).

 © Th. Worsch 2000-2015

2 Erste Beispiele

2.1 Ein randomisierter Identitätstest

Im folgenden geht es um die Aufgabe, die Gleichheit zweier „großer“ Objekte zu überprüfen. Dabei werden die Objekte nicht in ihrer gesamten Struktur verglichen, sondern es werden eine Art „Fingerabdrücke“ der Objekte berechnet und miteinander verglichen.

Im Unterschied zur Anwendung in der Kriminalistik kann man aber nur sicher sein, dass verschiedene Fingerabdrücke von verschiedenen Objekten stammen. Gleichheit von Fingerabdrücken wird zwar auch zum Anlass genommen, die Gleichheit der Objekte zu vermuten, dies kann aber (zum Beispiel mit einer Wahrscheinlichkeit von 1/4) falsch sein.

2.1 Gegeben seien drei $n \times n$ Matrizen $\mathbf{A}$, $\mathbf{B}$ und $\mathbf{C}$. Die Aufgabe bestehe darin, festzustellen, ob $\mathbf{AB} = \mathbf{C}$ ist.

Die Einträge der Matrizen mögen aus einem Körper $\mathbb{F}$ stammen. Dessen neutrale Elemente bzgl. Addition bzw. Multiplikation seien wie üblich mit 0 resp. 1 bezeichnet.

2.2 Die Aufgabe ist natürlich ganz einfach zu lösen, indem man $\mathbf{AB}$ ausmultipliziert und das Ergebnis mit $\mathbf{C}$ vergleicht. Dafür benötigt man bei Benutzung des besten derzeit bekannten Multiplikationsalgorithmus $\Theta(n^{2.376\cdots})$ Schritte (Coppersmith und Winograd 1990).

Wir wollen im folgenden eine probabilitische Methode von Freivalds (1977) betrachten, die in Zeit $O(n^2)$ mit Wahrscheinlichkeit $1 - 2^{-k}$ die richtige Antwort liefert.

2.3 Algorithmus.

⟨*Mit einem „Bit“ ist einer der Werte* 0 *oder* 1 *gemeint.*⟩
$\mathbf{r} \leftarrow$ ⟨*Vektor von* n *unabhängigen Zufallsbits*⟩
$\mathbf{x} \leftarrow \mathbf{Br}$
$\mathbf{y} \leftarrow \mathbf{Ax}$
$\mathbf{z} \leftarrow \mathbf{Cr}$
if $(\mathbf{y} \neq \mathbf{z})$ **then**
  return NO
else
  return YES
fi

Der Zeitbedarf dieses Algorithmus ist offensichtlich in $O(n^2)$.

2.4 Freivalds' Algorithmus überprüft also statt der Gleichheit $\mathbf{AB} = \mathbf{C}$ nur, ob $\mathbf{A}(\mathbf{Br}) = \mathbf{Cr}$ für einen zufällig gewählten Vektor $\mathbf{r}$ gilt. Die Werte $\mathbf{y} = \mathbf{A}(\mathbf{Br})$ bzw. $\mathbf{z} = \mathbf{Cr}$ werden sozusagen als *Fingerabdrücke* von $\mathbf{AB}$ resp. $\mathbf{C}$ benutzt.

2.5 Lemma. Es seien $\mathbf{A}$, $\mathbf{B}$ und $\mathbf{C}$ drei $n \times n$ Matrizen mit $\mathbf{AB} \neq \mathbf{C}$ und $\mathbf{r}$ sei ein Vektor von n Bits, die unabhängig und gleichverteilt gewählt wurden. Dann ist $\mathbf{Pr}[\mathbf{ABr} = \mathbf{Cr}] \leqslant 1/2$.

2.6 Beweis. Es sei $\mathbf{D} = \mathbf{AB} - \mathbf{C}$, also $\mathbf{D} \neq \mathbf{0}$. Außerdem setzen wir $\mathbf{y} = \mathbf{ABr}$ und $\mathbf{z} = \mathbf{Cr}$. Dann gilt $\mathbf{y} = \mathbf{z}$ genau dann, wenn $\mathbf{Dr} = \mathbf{0}$.

Es bezeichne $\mathbf{d}$ die erste Zeile von $\mathbf{D}$. O. B. d. A. sei $\mathbf{d}$ nicht der Nullvektor und die Nichtnulleinträge von $\mathbf{d}$ seien genau die ersten k Einträge $d_1, \dots, d_k$. (Diese Annahmen werden nur getroffen, um bequemer formulieren zu können. Man mache sich klar, dass der Beweis andernfalls im Prinzip analog geführt werden kann.)

Der erste Eintrag im Produktvektor $\mathbf{Dr}$ ist gerade $\mathbf{dr}$. Eine untere Schranke für die Wahrscheinlichkeit, dass dieser Eintrag nicht Null ist, ist auch eine untere Schranke für die Wahrscheinlichkeit, dass $\mathbf{Dr} \neq \mathbf{0}$ ist.

Wir suchen eine obere Schranke für die Wahrscheinlichkeit, dass $\mathbf{dr} = \mathbf{0}$ ist. Das ist genau dann der Fall, wenn $\sum_{i=1}^{k} d_i r_i = 0$, also genau dann, wenn $r_k = (-\sum_{i=1}^{k-1} d_i r_i)/d_k$. Um einzusehen, wie unwahrscheinlich dies ist, stelle man sich vor, $r_1, \dots, r_{k-1}$ seien bereits gewählt. Dann kann offensichtlich für höchstens *einen* der beiden möglichen Werte für r_k obige Gleichung richtig sein. Also ist diese Wahrscheinlichkeit höchstens $1/2$.

Mit einer Wahrscheinlichkeit größer gleich $1/2$ ist folglich $\mathbf{dr} \neq \mathbf{0}$ und damit auch $\mathbf{Dr} \neq \mathbf{0}$. ■

2.7 Korollar. Wenn $\mathbf{AB} = \mathbf{C}$ ist, liefert Algorithmus 2.3 stets die richtige Antwort. Wenn $\mathbf{AB} \neq \mathbf{C}$ ist, liefert Algorithmus 2.3 mit einer Wahrscheinlichkeit größer gleich $1/2$ die richtige Antwort.

2.8 Eine Fehlerwahrscheinlichkeit von $1/2$ ist sehr groß. Wie kann man sie kleiner machen?

Eine Möglichkeit findet man beim nochmaligen genauen Lesen des vorletzten Abschnittes von Beweis 2.6. Der Nenner 2 ergab sich nämlich aus der Tatsache, dass für die Komponenten von $\mathbf{r}$ jeweils zufällig einer von *zwei* Werten gewählt wurde. Enthält der Körper $\mathbb{F}$ aber mindestens 2^k Elemente, aus denen man gleichverteilt und unabhängig wählt, dann sinkt entsprechend die Fehlerwahrscheinlichkeit auf 2^{-k}.

Eine andere Möglichkeit, die Fehlerwahrscheinlichkeit zu drücken ist allgemeiner und auch bei anderen Gelegenheiten anwendbar: Man führt k unabhängige Wiederholung des Algorithmus durch und produziert nur dann am Ende die Antwort YES, wenn jeder Einzelversuch diese Antwort geliefert hat. Damit kann die Fehlerwahrscheinlichkeit von $1/2$ auf 2^{-k} (allgemein von p auf p^k) gedrückt werden.

Warum ist das so?

Es seien $Y_1, Y_2, \dots, Y_k$ die Zufallsvariablen für die Antworten bei den einzelnen Versuchen und es sei $\mathbf{AB} \neq \mathbf{C}$. Dann ist die Wahrscheinlichkeit für eine falsche Antwort des Algorithmus

$$\mathbf{Pr}\left[Y_1 = \text{YES} \land Y_2 = \text{YES} \land \cdots \land Y_k = \text{YES}\right] \quad \text{und das ist gleich} \quad \prod_{i=1}^{k} \mathbf{Pr}\left[Y_i = \text{YES}\right]$$

wegen der Unabhängigkeit der Y_i.

Wir merken noch an, dass bei beiden eben diskutierten Möglichkeiten zur Senkung der Fehlerwahrscheinlichkeit die gleiche Anzahl von Zufallsbits benötigt wird. Diese Zahl wächst proportional mit k. Alternativen, wie man das gleiche Ergebnisse erreichen kann, aber dabei unter Umständen deutlich weniger Zufallsbits benötigt, sind Gegenstand eines späteren Kapitels der Vorlesung.

 © Th. Worsch 2000-2015

2.2 Vergleich von Wörtern

Wir wollen noch zwei ähnlich gelagerte Probleme betrachten, bei denen es darauf ankommt, Wörter miteinander zu vergleichen. Der bequemeren Notation wegen werden wir uns auf Bitfolgen beschränken. Man mache sich aber als Übungsaufgabe klar, dass die Algorithmen und ihre Analysen für den Fall größerer Alphabete angepasst werden können.

Zunächst betrachten wir die wohl einfachste Aufgabe, den Test zweier Folgen auf Gleichheit.

2.9 Gegeben seien zwei „Datenbestände", die als gleich lange Bitfolgen (also Wörter) $a_1 \cdots a_n$ und $b_1 \cdots b_n$ repräsentiert sind.

Die Aufgabe besteht darin herauszufinden, ob die beiden Bitfolgen gleich sind. Interpretiert man sie auf die naheliegende Weise als Zahlen $a = \sum_{i=1}^n a_i 2^{i-1}$ und $b = \sum_{i=1}^n b_i 2^{i-1}$, muss also festgestellt werden, ob $a = b$ ist.

2.10 Man stelle sich zum Beispiel vor, dass a und b an verschiedenen Orten A und B vorliegen. Die Aufgabe bestehe darin, Gleichheit festzustellen, und dabei möglichst wenige Bits zu übertragen.

Es ist klar, dass man diese Aufgabe deterministisch lösen kann, indem man n Bits überträgt. Im folgenden soll ein probabilistischer Algorithmus analysiert werden, bei dem nur $O(\log n)$ Bits übertragen werden und der mit einer Fehlerwahrscheinlichkeit kleiner gleich $1/n$ die Aufgabe löst.

2.11 Algorithmus.

$\langle$*Überprüfung, ob Bitfolgen* $a_1 \cdots a_n$ *und* $b_1 \cdots b_n$ *gleich sind*$\rangle$
$p \leftarrow \langle$*Primzahl; zufällig gleichverteilt aus denen kleiner oder gleich* $n^2 \ln n^2$ *ausgewählt.*$\rangle$
$a \leftarrow \sum_{i=1}^n a_i 2^{i-1}$
$b \leftarrow \sum_{i=1}^n b_i 2^{i-1}$
if $(a \bmod p = b \bmod p)$ **then**
  return YES
else
  return NO
fi

Bevor wir den Algorithmus analysieren, seien zwei Dinge angemerkt. Die Aufgabe, ein geeignetes p auszuwählen, ist ebenfalls nicht ganz leicht. Überlegen Sie sich etwas!

Für eine Primzahl p bezeichne $F_p(x) : \mathbb{Z} \to \mathbb{Z}_p$ die Abbildung $x \mapsto x \bmod p$. Algorithmus 2.11 überprüft, ob $F_p(a) = F_p(b)$ ist, und liefert genau dann eine falsche Antwort, wenn das der Fall ist obwohl $a \neq b$ ist. Das ist genau dann der Fall, wenn p ein Teiler von $c = a - b$ ist.

Für die Anwendung des Algorithmus in dem in Punkt 2.10 skizzierten Szenario wird man z. B. am Ort A sowohl p „würfeln" als auch den Wert $F_p(a)$ berechnen und beides zu B übertragen, um dort $F_p(b)$ zu berechnen und mit $F_p(a)$ zu vergleichen. Da beide Werte kleiner oder gleich p sind, also kleiner als $n^2 \ln n^2$ ist, genügen dafür $O(\log n)$ Bits.

2.12 Satz. *Bei zufälliger gleichverteilter Wahl einer Primzahl* p *kleiner oder gleich* $n^2 \log n^2$ *ist*

$$\mathbf{Pr}\left[F_p(a) = F_p(b) \mid a \neq b\right] \in O(\frac{1}{n}) .$$

Für den Beweis benötigen wir ein schwieriges und ein leicht herzuleitendes Resultat.

 © Th. Worsch 2000-2015

2.13 SATZ. (CHEBYSHEV) *Für die Anzahl* $\pi(n)$ *der Primzahlen kleiner oder gleich* n *gilt:*

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

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

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

Der „Beweis" besteht in der Überlegung, dass jeder Primteiler größer oder gleich 2 ist, das Produkt von $n+1$ Primteilern also größer oder gleich 2^{n+1}.

2.15 BEWEIS. (VON SATZ 2.12) Algorithmus 2.11 liefert genau dann eine falsche Antwort, wenn $c \neq 0$ ist und von p geteilt wird. Da $c \leqslant 2^n$ ist, hat es nach dem vorangegangenen Lemma höchstens n verschiedene Primteiler.

Es sei t die obere Schranke des Intervalls, so dass der Algorithmus gleichverteilt eine der Primzahlen im Bereich $[2;t]$ auswählt. Nach Satz 2.13 gibt es dort $\pi(t) \in \Theta(t/\ln t)$ Primzahlen.

Die Wahrscheinlichkeit $\mathbf{Pr}[F_p(a) = F_p(b) \mid a \neq b]$, ein p zu wählen, das zu einer falschen Antwort führt, ist also höchstens $O(\frac{n}{t/\ln t})$.

Wählt man nun wie im Algorithmus $t = n^2 \ln n^2$, so ergibt sich eine obere Schranke in $O(1/n)$.

■

Wir kommen nun zu der zweiten Aufgabenstellung.

2.16 Beim sogenannten *pattern matching* sind ein *Text* $x = x_1 \cdots x_n$ und ein kürzeres *Suchmuster* $y = y_1 \cdots y_m$ gegeben. Die Aufgabe besteht darin, herauszufinden, ob für ein $1 \leqslant j \leqslant n-m+1$ gilt: Für alle $1 \leqslant i \leqslant m$ ist $y_i = x_{j+i-1}$.

Bezeichnet $x(j)$ das Teilwort $x_j \cdots x_{j+m-1}$ der Länge m von x, dann gilt es also herauszufinden, ob für ein $1 \leqslant j \leqslant n-m+1$ das zugehörige $x(j) = y$ ist.

Die letzte Formulierung lässt es schon naheliegend erscheinen, die zuvor untersuchte Technik erneut zu verwenden.

Zuvor sei darauf hingewiesen, dass man das Problem deterministisch in Zeit $O(m+n)$ lösen kann, etwa mit den Algorithmen von Boyer und J. S. Moore (1977) oder D. E. Knuth, Morris und Pratt (1977). Das ist besser als der triviale deterministische Algorithmus, der für jedes j alle Bits von $x(j)$ und y vergleicht und daher $O(nm)$ Zeit benötigt.

Im folgenden werden zwei randomisierte Algorithmen vorgestellt. Beim ersten handelt es sich um ein Monte-Carlo-Verfahren, beim zweiten um ein Las-Vegas-Verfahren, die Fehlerwahrscheinlichkeit ist dort also stets 0.

2.17 Die grundlegende Idee besteht wieder darin, statt y und ein $x(j)$ direkt zu vergleichen, dies mit

 © Th. Worsch 2000-2015

den Fingerabdrücken $F_p(y)$ und $F_p(x(j))$ zu tun. Eine wesentliche Beobachtung ist dabei:

$$\begin{aligned} x(j+1) &= \sum_{i=1}^{m} x_{j+1+i-1} 2^{i-1} \\ &= \frac{1}{2}(x_j - x_j) + \frac{1}{2}\sum_{i=1}^{m-1} x_{j+1+i-1} 2^{i} + x_{j+1+m-1} 2^{m-1} \\ &= \frac{1}{2}\left(\sum_{i=0}^{m-1} x_{j+i} 2^{i} - x_j\right) + x_{j+m} 2^{m-1} \\ &= \frac{1}{2}\left(\sum_{i=1}^{m} x_{j+i-1} 2^{i-1} - x_j\right) + x_{j+m} 2^{m-1} \\ &= \frac{1}{2}\left(x(j) - x_j\right) + x_{j+m} 2^{m-1} \ . \end{aligned}$$

Man kann also mit einer *konstanten* Anzahl von Operationen $x(j+1)$ aus $x(j)$ berechnen und folglich auch $F_p(x(j+1))$ aus $F_p(x(j))$. Der folgende Algorithmus benötigt daher $O(m+n)$ Schritte.

2.18 Algorithmus.

⟨*Überprüfung, ob Bitfolge* $y_1 \cdots y_m$ *in* $x_1 \cdots x_n$ *vorkommt*⟩
⟨*Ausgabe: erstes* j*, wo das der Fall ist, oder* -1 *sonst.*⟩
$p \leftarrow$ ⟨*Primzahl; zufällig gleichverteilt aus denen kleiner oder gleich* $n^2 m \ln n^2 m$ *ausgewählt.*⟩
$y \leftarrow \sum_{i=1}^{m} y_i 2^{i-1} \bmod p$
$z \leftarrow \sum_{i=1}^{m} x_i 2^{i-1} \bmod p$
for $j \leftarrow 1$ **to** $n-m$ **do**
  if $(y = z)$ **then**
    return j ⟨*die erste Stelle, an der eine „Übereinstimmung" gefunden wurde*⟩
  fi
  $z \leftarrow (z - x_j)/2 + x_{j+m} 2^{m-1} \bmod p$
od
return -1 ⟨y *kommt sicher nicht in* x *vor*⟩

2.19 Satz. *Algorithmus 2.18 liefert höchstens mit Wahrscheinlichkeit* $O(1/n)$ *eine falsche Antwort.*

2.20 Beweis. Ähnlich wie in Beweis 2.15 sei t die obere Schranke des Intervalls, so dass der Algorithmus gleichverteilt eine der Primzahlen im Bereich $[2;t]$ auswählt. Nach Satz 2.13 gibt es dort $\pi(t) \in \Theta(t/\ln t)$ Primzahlen.

Die Wahrscheinlichkeit $\mathbf{Pr}[F_p(y) = F_p(x(j)) \mid y \neq x(j)]$, ein p zu wählen, das an der Stelle j zu einer falschen Antwort führt, ist höchstens $O(\frac{m}{t/\ln t})$ (da $|y - x(j)|$ höchstens m verschiedene Primteiler besitzt). Die Wahrscheinlichkeit, dass eine falsche Antwort gegeben wird, weil an irgendeiner der $O(n)$ Stellen der Test versagt, ist daher höchstens $O(\frac{nm}{t/\ln t})$.

Wählt man nun wie im Algorithmus $t = n^2 m \ln n^2 m$, so ergibt sich eine obere Schranke in $O(1/n)$. ∎

© Th. Worsch 2000-2015

2.3 Ein randomisierter Quicksortalgorithmus

Der folgende Algorithmus ist eine naheliegende randomisierte Variante eines einfachen Quicksort-Algorithmus. Der Einfachheit halber nehmen wir an, dass alle Eingaben paarweise verschieden sind.

2.21 ÜBUNG. Wie kann man jeden Algorithmus, der voraussetzt, dass alle Eingaben paarweise verschieden sind, so modifizieren, dass er auch dann richtig arbeitet, wenn mehrere Elemente mit gleichem Sortierschlüssel auftreten? (Und zwar sogar so, dass die relative Reihenfolge solcher Elemente erhalten bleibt!)

2.22 ALGORITHMUS.
proc $R[1 \ldots n] \leftarrow \text{RandQuickSort}(S[1:n])$
$\langle$*Eingabe: ein Feld* $S[1:n]$ *paarweise verschiedener Zahlen*$\rangle$
$\langle$*Ausgabe: ein Feld* $R[1:n]$ *der Zahlen aus* S *in sortierter Reihenfolge*$\rangle$
$\langle$*Zwischenablage in Feldern* S_1 *und* $S_2$$\rangle$
$i \leftarrow \textbf{random}(1, n)$ $\langle$*liefere gleichverteilt eine natürliche Zahl zwischen* 1 *und* $n$$\rangle$
$y \leftarrow S[i]$
$j_1 \leftarrow 1; j_2 \leftarrow 1;$
for $i \leftarrow 1$ **to** n **do**
  if $S[i] < y$ **then** $S_1[j_1] \leftarrow S[i]; j_1 \leftarrow j_1 + 1$ **fi**
  if $S[i] > y$ **then** $S_2[j_2] \leftarrow S[i]; j_2 \leftarrow j_2 + 1$ **fi**
od
return $\text{RandQuickSort}(S_1[1:j_1-1]) \cdot y \cdot \text{RandQuickSort}(S_2[1:j_2-1])$ $\langle \cdot$ *sei Konkatenation*$\rangle$

2.23 Der obige Algorithmus liefert unabhängig davon, welcher Wert auf einer Rekursionsstufe als Pivotelement gewählt wird, stets die korrekte Ausgabe.

Allerdings haben die Auswahlen der y einen Einfluß auf die Laufzeit des Algorithmus (selbst bei festgehaltener Eingabefolge). Wird zum Beispiel immer der kleinste vorhandene Wert als Pivotelement gewählt, so ergibt sich eine quadratische Laufzeit. Wird dagegen immer der Median gewählt, ist die Laufzeit $O(n \log n)$.

Die Laufzeit dieses Algorithmus ist also eine Zufallsvariable. Im folgenden soll ihr *Erwartungswert* bestimmt werden.

Als erstes sieht man, dass er nur von der Anzahl n zu sortierender Elemente abhängt, aber nicht von den konkreten Werten. Des weiteren wollen wir den Erwartungswert der Laufzeit nur bis auf einen konstanten Faktor bestimmen. Deswegen genügt es, festzustellen, wie oft je zwei Feldelemente miteinander verglichen werden, denn zwei Feldelemente werden höchstens einmal miteinander verglichen.

2.24 Es bezeichne X_{ij} die Zufallsvariable, die angibt, ob die Elemente $R[i]$ und $R[j]$ miteinander verglichen wurden ($X_{ij} = 1$) oder nicht ($X_{ij} = 0$). Gesucht ist dann also

$$\mathbf{E}\left[\sum_{i=1}^{n} \sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j>i} \mathbf{E}\left[X_{ij}\right]$$

Ist p_{ij} die Wahrscheinlichkeit für den Vergleich von $R[i]$ und $R[j]$, so ist $\mathbf{E}\left[X_{ij}\right] = 1 \cdot p_{ij} + 0 \cdot (1 - p_{ij}) = p_{ij}$. Das Problem besteht also im wesentlichen darin, die p_{ij} zu bestimmen.

 © Th. Worsch 2000-2015

2.25 Man beachte, dass diese Wahrscheinlichkeiten *nicht* alle gleich sind, und überlege sich, warum das so ist!

2.26 SATZ. *Der Erwartungswert der Laufzeit von* RandQuickSort *für Eingaben der Länge* n *ist in* $O(n \log n)$.

2.27 BEWEIS. Es seien nun ein i und ein j ($1 \leqslant i < j \leqslant n$) beliebig aber fest vorgegeben. Zur Bestimmung von p_{ij} betrachtet man binäre Bäume mit den zu sortierenden Zahlen als Knoten, die durch je eine Ausführung von RandQuickSort wie folgt rekursiv festgelegt sind: Die Wurzel des Baumes ist das zufällig gewählte Pivotelement y. Der linke Teilbaum ergibt sich rekursiv nach der gleichen Regel gemäß der Ausführung beim Aufruf RandQuickSort$(S_1[1 : j_1 - 1])$ und analog der rechte gemäß der Ausführung beim Aufruf RandQuickSort$(S_2[1 : j_2 - 1])$. Bei einem In-Order-Durchlauf des Baumes ergäbe sich also die sortierte Reihenfolge der Werte.

Von Interesse ist im Folgenden aber eine andere Reihenfolge: Beginnend bei der Wurzel werden nacheinander absteigend auf jedem Niveau jeweils von links nach rechts alle Knoten besucht.

(Auch) bei dieser Besuchsreihenfolge wird irgendwann zu ersten Mal ein Element $R[k]$ angetroffen, für das $i \leqslant k \leqslant j$ ist. Es ist bei den Vergleichen in diesem Rekursionsschritt, dass tatsächlich entschieden wird, dass $R[i]$ *vor* $R[j]$ einzusortieren ist.

Dabei gibt es zwei Fälle:

1. Es ist $k = i$ oder $k = j$, d. h. eines der Elemente $R[i]$ und $R[j]$ ist das Pivotelement und die beiden werden miteinander verglichen.

2. Es ist $i < k < j$, d. h. ein anderes Element ist Pivot, $R[i]$ und $R[j]$ werden nicht miteinander verglichen, kommen in verschiedene Teilbäume (da auch $R[i] < R[k] < R[j]$ ist) und werden folglich auch später nie mehr miteinander verglichen.

Nun wird im Algorithmus jedes (noch) zur Verfügung stehende Element gleichwahrscheinlich als Pivotelement ausgewählt. Außerdem wissen wir, dass offensichtlich eines der $j - i + 1$ Elemente $R[i], \dots, R[j]$ ausgewählt wurde.

Im betrachteten Rekursionsschritt wird also eines von $j - i + 1$ Elementen gleichwahrscheinlich ausgewählt, und in genau zwei dieser Fälle ($k = i$ und $k = j$) werden $R[i]$ und $R[j]$ miteinander verglichen.

Also ist die Wahrscheinlichkeit dafür, dass $R[i]$ und $R[j]$ überhaupt miteinander verglichen werden gerade $p_{ij} = 2/(j - i + 1)$.

Damit können wir nun abschätzen:

$$\mathbf{E}\left[\sum_{i=1}^{n-1} \sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j>i} \mathbf{E}\left[X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \leqslant 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} .$$

Nun ist $H_n = \sum_{k=1}^{n} \frac{1}{k}$ die sogenannte n-te harmonische Zahl, für die man weiß: $H_n = \ln n + \Theta(1)$. Hieraus folgt die Behauptung. ∎

2.28 Es sei auf einen Unterschied zwischen den beiden in diesem Kapitel behandelten Algorithmen hingewiesen: Der randomisierte Quicksortalgorithmus liefert *immer* das richtige Ergebnis. Beim Identitätstest nimmt man dagegen in Kauf, dass die Antwort manchmal nicht zutreffend ist, allerdings so „selten“, dass man die Fehlerwahrscheinlichkeit leicht beliebig weit drücken kann.

Außerdem ist letzterer ein Entscheidungsproblem, für das nur zwei Antworten in Frage kommen.

 © Th. Worsch 2000-2015

2.29 Arbeiten über randomisierte Primzahltests (Rabin 1976; Solovay und Strassen 1977; Solovay und Strassen 1978) gehören zu den Klassikern. Auch diese Algorithmen liefern manchmal die falsche Antwort, aber immer nur für zusammengesetzte Zahlen die Behauptung „prim"; für Primzahlen ist die Antwort stets richtig.

Schwieriger war es für L. M. Adleman und Huang (1987), einen randomisierten Algorithmus anzugeben, der für Primzahlen manchmal fälschlicherweise behauptet sie sei zusammengesetzt, aber für zusammengesetzte Zahlen stets die richtige Antwort liefert.

Vor einigen Jahren haben schließlich Agrawal, Kayal und Saxena (2002) bewiesen, dass das Problem sogar deterministisch in Polynomialzeit gelöst werden kann.

Zusammenfassung

1. Randomisierte Algorithmen enthalten eine Zufallskomponente.
2. Das führt im Allgemeinen dazu, dass — *bei festgehaltener Eingabe* — z. B. die Laufzeit eine randomisierten Algorithmus eine Zufallsvariable ist.
3. Manche randomisierten Algorithmen liefern immer das richtige Ergebnis.
4. Manche randomisierten Algorithmen liefern unter Umständen ein falsches Ergebnis. Dann ist man im Allgemeinen an kleinen Fehlerwahrscheinlichkeiten interessiert.

Literatur

Adleman, L. M. und A. M.-D. Huang (1987). „Recognizing Primes in Random Polynomial Time". In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. S. 462–469 (siehe S. 17, 20).

Agrawal, Manindra, Neeraj Kayal und Nitin Saxena (2002). *PRIMES is in P*. Report. Kanpur-208016, India: Department of Computer Science und Engineering, Indian Institute of Technology Kanpur. URL: http://www.cse.iitk.ac.in/news/primality.pdf (siehe S. 17, 20).

Boyer, Robert S. und J. Strother Moore (1977). „A fast string search algorithm". In: *Communications of the ACM* 20.10, S. 762–772 (siehe S. 13).

Coppersmith, Don und Shmuel Winograd (1990). „Matrix Multiplication via Arithmetic Progressions". In: *Journal of Symbolic Computation* 9.3, S. 251–280 (siehe S. 10).

Freivalds, Rusins (1977). „Probabilistic machines can use less running time". In: *Information Processing 77, Proceedings of the IFIP Congress 1977*. Hrsg. von B. Gilchrist. Amsterdam: North Holland, S. 839–842 (siehe S. 10).

Knuth, D. E., J. H. Morris und V. R. Pratt (1977). „Fast pattern matching in strings". In: *SIAM Journal on Computing* 6.2, S. 323–350 (siehe S. 13).

Rabin, Michael O. (1976). „Probabilistic Algorithms". In: *Algorithms and Complexity*. Hrsg. von J. F. Traub. Academic Press, S. 21–39 (siehe S. 7, 17, 20).

Solovay, R. und V. Strassen (1977). „A Fast Monte-Carlo Test for Primality". In: *SIAM Journal on Computing* 6.1, S. 84–85 (siehe S. 7, 17).

 © Th. Worsch 2000-2015

— (1978). „Erratum: A Fast Monte-Carlo Test for Primality“. In: *SIAM Journal on Computing* 7.1, S. 118 (siehe S. 7, 17).

 © Th. Worsch 2000-2015

3 Probabilistische Komplexitätsklassen

3.1 Probabilistische Turingmaschinen

3.1 Wir gehen davon aus, dass die Konzepte deterministischer und nichtdeterministischer Turingmaschinen im wesentlichen bekannt sind.

Die Menge der („echten“, s.u.) Zustände wird mit S bezeichnet und das Bandalphabet mit B. Das Blanksymbol wird gegebenenfalls als $\square$ geschrieben. $A \subseteq B - \{\square\}$ ist das Eingabealphabet.

Wir beschränken uns auf TM mit einem Band und einem Kopf. In diesem Fall ist bei einer deterministischen TM (DTM) die Überführungsfunktion von der Form $\delta : S \times B \to (S \cup \{\text{YES}, \text{NO}\}) \times B \times \{-1, 0, 1\}$. Für die Endzustände YES und NO müssen keine Regeln angegeben werden.

Eingaben werden auf dem ansonsten mit $\square$-Symbolen beschrifteten Band zur Verfügung gestellt, wobei sich der Kopf anfangs auf dem ersten Eingabesymbol befindet. Wir beschränken uns (weitgehend) auf Entscheidungsprobleme (i. e. die Erkennung formaler Sprachen). Dabei wird das Ergebnis (Annahme oder Ablehnung) am erreichten Endzustand (YES oder NO) abgelesen.

Für eine nichtdeterministische TM (NTM) ist die Überführungsfunktion von der Form $\delta : S \times B \to 2^{(S \cup \{\text{YES},\text{NO}\}) \times B \times \{-1,0,1\}}$.

Die von einer TM T *akzeptierte Sprache* ist

$$L(T) = \{w \in A^+ \mid \text{ es gibt eine akzeptierende Berechnung von } T \text{ für Eingabe } w\} .$$

Wie schon zu Beginn des ersten Kapitels erwähnt, kann man für probabilistische Turingmaschinen formal unterschiedliche Definitionen geben, die aber inhaltlich äquivalent sind.

3.2 DEFINITION Eine *probabilistische Turingmaschine* (PTM) ist formal wie eine nichtdeterministische TM festgelegt, jedoch mit der zusätzlichen Einschränkung, dass für alle Paare (s, b) genau eine oder zwei mögliche Alternativen für den nächsten Schritt vorliegen: $1 \leqslant |\delta(s, b)| \leqslant 2$. Und es wird vereinbart, dass eine PTM in einer Berechnungssituation, in der zwei Möglichkeiten vorliegen, jede mit gleicher Wahrscheinlichkeit 1/2 wählt. ◇

3.3 Diese „lokale“ Quantifizierung zusammen mit der anderen Sichtweise, sich z. B. für die Wahrscheinlichkeit des Akzeptierens einer Eingabe (also quantitative Eigenschaften des globalen Verhaltens) zu interessieren, macht den wesentlichen Unterschied zu nichtdeterministischen TM aus.

Man beachte auch, dass 1/2 ein „harmloser“ Wert ist. Es ist nicht erlaubt, andere Wahrscheinlichkeiten p zu wählen, bei denen etwa in die Dezimalbruchentwicklung von p die Lösung des Halteproblems hineinkodiert ist.

3.4 DEFINITION Wir wollen sagen, dass eine PTM in *Normalform* vorliege, wenn sie eine PTM ist, die die folgenden Bedingungen erfüllt:

- Für jedes Paar $(s, b) \in S \times B$ gibt es *genau* zwei mögliche Alternativen: $|\delta(s, b)| = 2$.

- Für jede Eingabe sind alle Berechnungen gleich lang. ◇

3.5 Für PTM in Normalform bereitet es dann auch keine Schwierigkeiten die *Zeitkomplexität* $t(n)$ auf die naheliegende Weise zu definieren. Eine TM arbeite in *Polynomialzeit*, wenn es ein Polynom $p(n)$ gibt, so dass $t(n) \leqslant p(n)$ ist.

Der Berechnungsbaum einer PTM in Normalform ist also für jede Eingabe eine voller balancierter binärer Baum.

3.2 Komplexitätsklassen

3.6 Wir benutzen die folgenden üblichen Abkürzungen für wichtige Komplexitätsklassen:

Klasse $\mathcal{C}$	Kriterium für $L \in \mathcal{C}$
P	L kann von einer DTM in Polynomialzeit erkannt werden
NP	L kann von einer NTM in Polynomialzeit erkannt werden
PSPACE	L kann von einer DTM oder NTM in polynomialem Platz erkannt werden

Für eine Komplexitätsklasse $\mathcal{C}$ sei co$-\mathcal{C} = \{L \mid \bar{L} \in \mathcal{C}\}$. Bekanntlich ist $\mathbf{P} = $ co$-\mathbf{P}$ und $\mathbf{P} \subseteq \mathbf{NP} \cap$ co$-\mathbf{NP}$. Die Beziehung zwischen **NP** und co$-$**NP** ist ungeklärt. Man weiß, dass $\mathbf{NP} \cup$ co$-\mathbf{NP} \subseteq \mathbf{PSPACE}$ ist.

Eine *Polynomialzeitreduktion* einer Sprache L_1 auf eine Sprache L_2 ist eine Abbildung $f : A^+ \to A^+$, die in Polynomialzeit berechnet werden kann und für die für alle $w \in A^+$ gilt:

$$w \in L_1 \iff f(w) \in L_2 \ .$$

In einer solchen Situation folgt z. B. aus $L_2 \in \mathbf{NP}$ sofort auch $L_1 \in \mathbf{NP}$.

Eine Sprache H ist **NP**-*hart*, wenn jede Sprache aus **NP** auf H polynomialzeitreduziert werden kann. Eine Sprache ist **NP**-*vollständig*, wenn sie **NP**-hart und aus **NP** ist.

Im Hinblick auf die nachfolgenden Definitionen randomisierter Komplexitätsklassen schreiben wir noch einmal die für **P** etwas anders auf. Es ist genau dann $L \in \mathbf{P}$, wenn es eine DTM T gibt, die in Polynomialzeit arbeitet und für die für alle Eingaben $w \in A^+$ gilt:

- $w \in L \implies$ T akzeptiert w
- $w \notin L \implies$ T akzeptiert w nicht

Wir definieren nun einige wichtige randomisierte Komplexitätsklassen.

3.7 Definition Im folgenden sind alle Implikationen als für alle $w \in A^+$ quantifiziert zu verstehen.

1. $L \in \mathbf{RP}$, wenn es eine Polynomialzeit-PTM T gibt, für die gilt:
   - $w \in L \implies \mathbf{Pr}\,[\text{T akzeptiert } w] \geqslant 1/2$
   - $w \notin L \implies \mathbf{Pr}\,[\text{T akzeptiert } w] = 0$
2. $\mathbf{ZPP} = \mathbf{RP} \cap$ co$-\mathbf{RP}$.
3. $L \in \mathbf{BPP}$, wenn es eine Polynomialzeit-PTM T gibt, für die gilt:

 © Th. Worsch 2000-2015

- $w \in L \implies \mathbf{Pr}[\mathrm{T} \text{ akzeptiert } w] > 3/4$
- $w \notin L \implies \mathbf{Pr}[\mathrm{T} \text{ akzeptiert } w] < 1/4$

4. $L \in \mathbf{PP}$, wenn es eine Polynomialzeit-PTM T gibt, für die gilt:

   - $w \in L \implies \mathbf{Pr}[\mathrm{T} \text{ akzeptiert } w] > 1/2$
   - $w \notin L \implies \mathbf{Pr}[\mathrm{T} \text{ akzeptiert } w] \leqslant 1/2$ ◇

Im Falle von **BPP** ist also die Fehlerwahrscheinlichkeit in beiden Fällen ($w \in L$ bzw. $w \notin L$) kleiner als $1/4$.

Man kann zeigen, dass man bei der Definition von **PP** bei $w \notin L$ statt $\leqslant 1/2$ auch $< 1/2$ hätte schreiben können, ohne inhaltlich etwas zu ändern. Insofern darf man sagen, dass im Falle von **PP** die Fehlerwahrscheinlichkeit kleiner $1/2$ ist.

3.8 Bei **RP** besteht die Möglichkeit *einseitiger* Fehler, bei **BPP** und **PP** die *zweiseitiger* Fehler. Wir werden sehen, dass man im Falle von **ZPP** Maschinen konstruieren kann, deren Antworten nie fehlerhaft sind.

Generell bezeichnet man randomisierte Algorithmen, die eine endliche erwartete Laufzeit haben und deren Antworten nie falsch sind, als *Las Vegas*-Algorithmen. Algorithmen, die möglicherweise fehlerhafte Antworten liefern, heißen auch *Monte Carlo*-Algorithmen.

3.9 Beispiel. Der Primzahltest von Rabin (1976) zeigt, dass die Sprache der Primzahlen in co−**RP** liegt. Viel schwieriger war es, nachzuweisen, dass sie auch in **RP** (und damit in **ZPP**) ist (L. M. Adleman und Huang 1987). Inzwischen weiß man, dass PRIMES sogar in **P** liegt (Agrawal, Kayal und Saxena 2002).

3.10 Für PTM, die in Normalform sind, kann man die Klassen **RP**, **BPP** und **PP** auch alternativ wie folgt festlegen:

1. $L \in \mathbf{RP}$, wenn es eine Polynomialzeit-PTM T gibt, für die gilt:
   - $w \in L \implies$ mindestens $1/2$ aller möglichen Berechnungen liefern Antwort yes.
   - $w \notin L \implies$ alle Berechnungen liefern Antwort no.
2. $L \in \mathbf{BPP}$, wenn es eine Polynomialzeit-PTM T gibt, für die gilt:
   - $w \in L \implies$ mindestens $3/4$ aller möglichen Berechnungen liefern Antwort yes.
   - $w \notin L \implies$ mindestens $3/4$ aller möglichen Berechnungen liefern Antwort no.
3. $L \in \mathbf{PP}$, wenn es eine Polynomialzeit-PTM T gibt, für die gilt:
   - $w \in L \implies$ mehr als die Hälfte aller möglichen Berechnungen liefern Antwort yes.
   - $w \notin L \implies$ mindestens die Hälfte aller möglichen Berechnungen liefern Antwort no.

Eine PTM, die die Zugehörigkeit der von ihr erkannten Sprache zu **RP** (resp. **BPP** oder **PP**) belegt, wollen wir auch eine **RP**-PTM (resp. **BPP**-PTM oder **PP**-PTM) nennen.

Im Fall von **PP** wird die Entscheidung sozusagen durch absolute Mehrheit getroffen.

3.11 Die Definition von **RP** beinhaltet ein Problem. Es ist für eine vorgegebene PTM (gleichgültig, ob in Normalform oder nicht) nicht ersichtlich, ob sie ein Beweis dafür ist, dass die von ihr erkannte Sprache in **RP** ist. Das ist sogar *unentscheidbar*.

Die analoge Aussage gilt auch für **BPP**.

 © Th. Worsch 2000-2015

3.12 Satz. *Für jedes Polynom $q(n) \geqslant 1$ kann man in der Definition von* **RP** *anstelle von $1/2$ auch $1-2^{-q(n)}$ einsetzen, also Fehlerwahrscheinlichkeit $2^{-q(n)}$ statt $1/2$ fordern, ohne an der Klasse etwas zu verändern.*

3.13 Beweis. Es sei R eine **RP**-PTM, die L mit Fehlerwahrscheinlichkeit $1/2$ erkennt. Eine **RP**-PTM R', die L mit Fehlerwahrscheinlichkeit $2^{-q(n)}$ erkennt, kann wie folgt konstruiert werden.

Es sei w eine Eingabe der Länge n. R' verwaltet einen Zähler, der mit $q(n)$ initialisiert wird. In einer Schleife wird er auf Null heruntergezählt und dabei jedes Mal eine Berechnung von R für die Eingabe w simuliert. Wenn R bei mindestens einem Versuch YES liefern würde, beendet R' seine Berechnung mit Antwort YES. Andernfalls antwortet R' mit NO.

Ist nun $w \in L(R)$, dann ist die Wahrscheinlichkeit, dass R' die falsche Antwort liefert, $(1/2)^{q(n)} = 2^{-q(n)}$. Ist $w \notin L(R)$, dann antwortet R und daher auch R' immer mit NO.

Bezeichnet $p(n)$ die Laufzeit von R, so ist die von R' gerade $p(n)q(n)$, also ebenfalls polynomiell. ■

3.14 Satz. *Für jedes Polynom $q(n) \geqslant 2$ kann man in der Definition von* **BPP** *anstelle der Fehlerwahrscheinlichkeit von $1/4$ auch $2^{-q(n)}$ einsetzen, ohne an der Klasse etwas zu verändern.*

3.15 Beweis. Es sei R eine **BPP**-PTM, die L mit Fehlerwahrscheinlichkeit $1/4$ erkennt. Eine **BPP**-PTM R', die L mit Fehlerwahrscheinlichkeit $2^{-q(n)}$ erkennt, kann wie folgt konstruiert werden.

Es sei w eine Eingabe der Länge n. Es sei $m = 2q + 1$ eine ungerade Zahl, die weiter unten geeignet festlegt wird.

R' verwaltet einen Zähler, der mit m (einer ungeraden Zahl) initialisiert wird. In einer Schleife wird er auf Null heruntergezählt und dabei jedes Mal eine Berechnung von R für die Eingabe w simuliert. Es wird gezählt, wie oft R Antwort YES liefern würde, und wie oft Antwort NO.

Die Antwort von R' ist die, die von R häufiger gegeben wurde.

Wir rechnen nun nach, dass R' die gewünschten Eigenschaften hat, wenn q geeignet gewählt wird.

Die Wahrscheinlichkeit, eine falsche Antwort zu erhalten, ist höchstens

$$\begin{aligned}
\sum_{i=0}^{q} \binom{m}{i} \left(\frac{3}{4}\right)^{i} \left(\frac{1}{4}\right)^{m-i} &\leqslant \sum_{i=0}^{q} \binom{m}{i} \left(\frac{3}{4}\right)^{i} \left(\frac{1}{4}\right)^{m-i} \left(\frac{3/4}{1/4}\right)^{m/2-i} \\
&= \sum_{i=0}^{q} \binom{m}{i} \left(\frac{3}{4}\right)^{m/2} \left(\frac{1}{4}\right)^{m/2} \\
&= \left(\frac{3}{16}\right)^{m/2} \sum_{i=0}^{q} \binom{m}{i} \\
&\leqslant \left(\frac{3}{16}\right)^{m/2} 2^{m-1} = \frac{1}{2}\left(\frac{3}{4}\right)^{m/2}
\end{aligned}$$

Die Wahrscheinlichkeit für eine richtige Antwort ist also größer gleich $1 - \frac{1}{2}\left(\frac{3}{4}\right)^{m/2}$. Man wählt nun m so, dass gilt (log sei zur Basis 2 genommen):

© Th. Worsch 2000-2015

$$\begin{aligned}
& 1-\frac{1}{2}\left(\frac{3}{4}\right)^{m/2} \geqslant 1-2^{-q(n)} \\
\iff\quad & \frac{1}{2}\left(\frac{3}{4}\right)^{m/2} \leqslant 2^{-q(n)} \\
\iff\quad & 2\left(\frac{4}{3}\right)^{m/2} \geqslant 2^{q(n)} \\
\iff\quad & \left(\frac{4}{3}\right)^{m/2} \geqslant 2^{q(n)-1} \\
\iff\quad & \frac{m}{2}(2-\log 3) \geqslant q(n)-1 \\
\iff\quad & m \geqslant \frac{2(q(n)-1)}{2-\log 3}
\end{aligned}$$

Wie man sieht, kann eine geeignete Anzahl von Wiederholungen linear in $q(n)$ gewählt werden, so dass mit der Laufzeit von R auch die von R' polynomiell ist. ■

3.16 SATZ. *Wenn* $L \in$ **ZPP** *ist, dann gibt es eine PTM, für die der Erwartungswert der Laufzeit polynomiell ist und die* L *entscheidet (d. h.* immer *richtige Antworten liefert).*

3.17 BEWEIS. Es sei R eine **RP**-PTM und $\bar{R}$ eine **RP**-PTM für $\bar{L}$, die bei jedem Lauf für eine Eingabe w als Antwort liefern. Der folgende Algorithmus leistet das Gewünschte:

⟨*Eingabe: Wort* w⟩
⟨*Ausgabe:* YES *falls* $w \in L$, NO *falls* $w \notin L$⟩
repeat
  $r \leftarrow R(w)$
  $r' \leftarrow$ **not** $\bar{R}(w)$ ⟨$\bar{R} =$ YES $\Longrightarrow w \in \bar{L}$, *also* $w \notin L$⟩
until $r = r'$
return r

Dass die Antwort dieses Algorithmus *immer* korrekt ist, ergibt sich aus der Tatsache, dass im Fall $w \in L$ (bzw. $w \notin L$, i. e. $w \in \bar{L}$) die Behauptung von $\bar{R}$ (bzw. R) garantiert richtig ist.

Es sei $p(n)$ das Maximum der (polynomiellen) Laufzeiten von R bzw. $\bar{R}$. Der Erwartungswert für die Laufzeit des obigen Algorithmus ist nach oben beschränkt durch

$$\frac{1}{2}2p(n)+\frac{1}{4}4p(n)+\frac{1}{8}6p(n)+\cdots = 2p(n)\sum_{i=1}^{\infty}2^{-i}i = 4p(n)\ .$$

(Hinweis: $2-\sum_{i=0}^{k} i2^{-i} = (k+2)2^{-k}$.) ■

3.18 Bei obigem Algorithmus kann es für eine Eingabe passieren, dass *nie* ein Ergebnis geliefert wird.

Deshalb treffen manche Autoren (z. B. Goos 1999) bei Las Vegas-Algorithmen noch die Unterscheidung, ob sie immer terminieren oder nicht. Im ersteren Fall sprechen sie dann genauer von *Macao*-Algorithmen.

 © Th. Worsch 2000-2015

3.3 Beziehungen zwischen Komplexitätsklassen

3.19 Gegenstand dieses Abschnitts sind die in Abbildung 3.1 dargestellten Inklusionsbeziehungen. Ein Pfeil von **A** nach **B** bedeutet, dass $\mathbf{A} \subseteq \mathbf{B}$ ist. Inklusionsbeziehungen, die sich durch Transitivität ergeben, sind nicht dargestellt. Ansonsten bedeutet die Abwesenheit eines Pfeiles in diesem Diagramm, dass man nicht weiß, ob eine Inklusionsbeziehung besteht oder nicht.

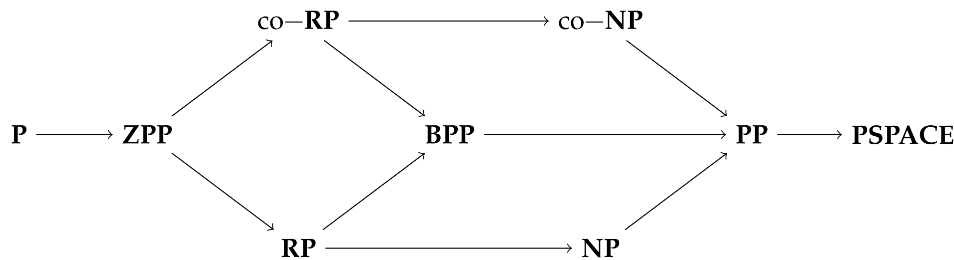


Abbildung 3.1: Beziehungen zwischen Komplexitätsklassen.

Da noch nicht einmal klar ist, ob die Inklusion $\mathbf{P} \subseteq \mathbf{PSPACE}$ echt ist oder nicht, weiß man das natürlich auch von keiner der „dazwischen liegenden“ Inklusionen.

Da jede deterministische TM gleichzeitig auch eine **RP**-PTM und eine co−**RP**-PTM ist, und aufgrund der Definition von **ZPP** ist zunächst einmal klar:

3.20 Satz. $\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP}$ *und* $\mathbf{ZPP} \subseteq \text{co}-\mathbf{RP}$.

Denkt man an die alternativen Beschreibungen der Komplexitätsklassen mit Hilfe von Normalform-PTM in Punkt 3.10, dann ist klar, dass gilt:

3.21 Satz. $\mathbf{RP} \subseteq \mathbf{NP}$ *und* $\text{co}-\mathbf{RP} \subseteq \text{co}-\mathbf{NP}$.

Nach Satz 3.12 darf man bei der Definition von **RP** (bzw. co−**RP**) ohne Beschränkung der Allgemeinheit eine Fehlerwahrscheinlichkeit von 1/4 einsetzen (man wähle $q(n) = 2$). Also gilt:

3.22 Satz. $\mathbf{RP} \subseteq \mathbf{BPP}$ *und* $\text{co}-\mathbf{RP} \subseteq \mathbf{BPP}$.

Offensichtlich gilt:

3.23 Satz. $\mathbf{BPP} \subseteq \mathbf{PP}$.

Für den Nachweis der beiden letzten Inklusionen muss man wieder etwas arbeiten.

3.24 Satz. $\mathbf{NP} \subseteq \mathbf{PP}$ *und* $\text{co}-\mathbf{NP} \subseteq \mathbf{PP}$.

3.25 Beweis. Wir beschränken uns auf den Nachweis der ersten Inklusion.

Es sei N eine Polynomialzeit-NTM in Normalform. Daraus wird eine PTM N' konstruiert, indem am Ende jeder Berechnung ein weiterer Schritt mit zwei Alternativen angehängt wird. Im einen Fall wird die ursprüngliche Antwort von N geliefert, im anderen Fall auf jeden Fall yes.

Ist eine Eingabe $w \in L(N)$, dann gibt es bei N mindestens eine akzeptierende Berechnung. Folglich ist bei N' mehr als die Hälfte aller Berechnungen akzeptierend. Ist dagegen $w \notin L(N)$, dann ist nur genau die Hälfte aller Berechnungen akzeptierend.

 © Th. Worsch 2000-2015

Da mit N offensichtlich auch N' in Polynomialzeit arbeitet, ist N' die gesuchte Maschine. ■

3.26 SATZ. **PP** $\subseteq$ **PSPACE**.

3.27 BEWEIS. Es sei P eine **PP**-PTM in Normalform mit Zeitkomplexität $p(n)$ und w eine beliebige Eingabe. Der folgende naheliegende deterministische Algorithmus leistet das Gewünschte:

```
⟨Eingabe: w⟩
a ← 0        ⟨für Zählung der akzeptierenden Berechnungen von P⟩
k ← p(|w|)        ⟨Anzahl Schritte von P⟩
⟨für alle Bitfolgen der Länge k⟩
for (b_k b_{k−1} ··· b_1) ← (000···0)  to  (111···1) do
    r ← Simulation von P(w) mit Entscheidungen gemäß den b_i
    if r = YES then a ← a+1 fi
od
if a > 2^{k−1} then
    return YES
else
    return NO
fi
```

Der Platzbedarf dieses Algorithmus wird dominiert von dem für die Bits b_i und dem für die Simulationen von $P(w)$. Die Anzahl k der Bits ist polynomiell in der Länge der Eingabe. Jede der Simulationen dauert polynomiell lange. In dieser Zeit kann daher auch nicht auf mehr als polynomiell viele Speicherplätze zugegriffen werden. Also ist der Gesamtplatzbedarf der obigen Prozedur polynomiell. ■

Zusammenfassung

1. Es gibt randomisierte Algorithmen ohne Fehler, mit einseitigem und mit zweiseitigem Fehler.

2. Die Fehlerwahrscheinlichkeit kann man relativ einfach reduzieren, benötigt dafür aber „deutlich" mehr Zufallsbits.

Literatur

Adleman, L. M. und A. M.-D. Huang (1987). „Recognizing Primes in Random Polynomial Time". In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. S. 462–469 (siehe S. 17, 20).

Agrawal, Manindra, Neeraj Kayal und Nitin Saxena (2002). *PRIMES is in P*. Report. Kanpur-208016, India: Department of Computer Science und Engineering, Indian Institute of Technology Kanpur. URL: http://www.cse.iitk.ac.in/news/primality.pdf (siehe S. 17, 20).

 © Th. Worsch 2000-2015

Goos, Gerhard (1999). *Vorlesung über Informatik*. Bd. 2. Heidelberg: Springer (siehe S. 22).
Rabin, Michael O. (1976). „Probabilistic Algorithms". In: *Algorithms and Complexity*. Hrsg. von J. F. Traub. Academic Press, S. 21–39 (siehe S. 7, 17, 20).

 © Th. Worsch 2000-2015

4 Routing in Hyperwürfeln

Anhand einer Problemstellung, die bei Parallelrechnern mit einer entsprechenden Verbindungsstruktur aufträte, sollen in diesem Kapitel zwei weitere Aspekte vertieft werden, die bei randomisierten Algorithmen bzw. ihrer Analyse von Bedeutung sind: Chernoff-Schranken und die probabilistische Methode.

4.1 Das Problem und ein deterministischer Algorithmus

4.1 Definition Für $d \geqslant 1$ ist der d-*dimensionale Hyperwürfel* H_d gegeben durch die Menge der Knoten $V_d = \{0, 1\}^d$ und die Menge der Kanten E_d, die zwei Knoten x und y genau dann miteinander verbinden, wenn sich die Bitfolgen x und y an genau einer Stelle unterscheiden. ◇

4.2 H_d hat also $N = 2^d$ Knoten und $d \cdot 2^{d-1} \in \Theta(N \log N)$ Kanten.

Der Durchmesser von H_d ist $d = \log N$. Ist nämlich $x = (x_1 x_2 \cdots x_d)$ und $y = (y_1 y_2 \cdots y_d)$ (mit $x_i, y_j \in \{0, 1\}$), so ergibt sich aus der Knotenfolge

$$(x_1 x_2 \cdots x_d), (y_1 x_2 \cdots x_d), \ldots, (y_1 y_2 \cdots y_{d-1} x_d), (y_1 y_2 \cdots y_d)$$

nach Entfernen aller aufeinander folgender doppelter Knoten ein Pfad von x nach y. Er hat maximal Länge d, da höchstens so viele Bits unterschiedlich sind.

4.3 Problem. Beim *Permutationsrouting* stellt man sich vor, die Knoten x von H_d seien Prozessoren, auf denen jeweils eine „Nachricht" (oder auch ein „Paket") vorliege, die auf einem Pfad in H_d zu einem Zielknoten $f(x)$ transportiert werden muss. Das Präfix „Permutation" bedeutet dabei, dass $f\colon V \to V$ eine Bijektion ist, also eine Permutation der Knoten beschreibt.

Es gibt die Einschränkung, dass in jedem Schritt über jede Kante maximal ein Paket transportiert werden kann. Entsteht die Situation, dass mehrere Pakete gleichzeitig über die gleiche Kante weitergeschickt werden sollen, so werde ein Paket bedient, während die anderen in einer FIFO-Warteschlange gespeichert und später bedient werden.

Die Aufgabe besteht darin, für jedes Paar $(x, f(x))$ einen „Reiseplan" (Kanten und Zeitpunkte für deren Benutzung) von x nach $f(x)$ festzulegen, so dass alle Pakete möglichst schnell ans Ziel kommen und dabei obiger Einschränkung Genüge getan wird.

4.4 Insbesondere wollen wir uns im folgenden nur für Algorithmen interessieren, die im Englischen als *oblivious* und im Deutschen (manchmal) als *datenunabhängig* bezeichnet werden. Damit ist gemeint, dass die Route für Paket x nicht von den Routen der anderen Pakete abhängt.

4.5 Die Argumentation in Punkt 4.2 für den Durchmesser von H_d liefert auch gleich ein Verfahren für die Routenwahl. Dieser „Bit-Fixing-Algorithmus" geht auf Valiant zurück.

4.6 Müsste man insgesamt nur *ein* Paket transportieren, so würden offensichtlich $d = \log N$ Schritte dafür ausreichen. Schwierig wird es dadurch, dass mehrere Pakete gleichzeitig transportiert werden müssen und sie sich unter Umständen gegenseitig behindern.

Für den Bit-Fixing-Algorithmus liefert zum Beispiel die folgende Permutation („Matrix-Transposition") sehr lange Transportzeiten: $f(x_1 \cdots x_{d/2} x_{d/2+1} \cdots x_d) = (x_{d/2+1} \cdots x_d x_1 \cdots x_{d/2})$. Der Grund ist schnell zu sehen: Für jedes beliebige Bitmuster $z_{d/2+1} \cdots z_d$ werden alle $2^{d/2} = \sqrt{N}$ Pakete, die in einem der Knoten $x_1 \cdots x_{d/2} z_{d/2+1} \cdots z_d$ starten, über den gleichen Knoten $z_{d/2+1} \cdots z_d z_{d/2+1} \cdots z_d$ geschickt. Da in jedem Schritt maximal d Pakete diesen Knoten verlassen können, ergibt sich für diesen Algorithmus eine untere Schranke von $\sqrt{N}/d$ Schritten.

Weniger offensichtlich ist, dass man diese Beobachtung verallgemeinern kann.

Die erste diesbezügliche Arbeit stammt von A. Borodin und Hopcroft (1985). Die nachfolgende Verschärfung geht im wesentlichen auf Kaklamanis, Krizanc und Tsantilas (1990) zurück.

4.7 Satz. *Zu jedem deterministischen datenunabhängigen Algorithmus für Permutationsrouting in einem Graphen mit N Knoten, die alle Ausgangsgrad d haben, gibt es eine Permutation, für die der Algorithmus $\Omega(\sqrt{N}/d)$ Schritte benötigt.*

4.8 Beweis. Der folgende Beweis stammt aus dem Buch von Leighton (1992).

Es sei A ein deterministischer datenunabhängiger Algorithmus für Permutationsrouting. Der von A für ein Paket von u nach v gewählte Pfad werde mit $P_{u,v}$ bezeichnet. A ist also durch die N^2 Pfade $P_{u,v}$ für alle Knoten $u, v \in V$ eindeutig charakterisiert.

Die Beweisidee besteht darin, „große" Mengen von Quellknoten $U' = \{u_1, \ldots, u_k\}$ und zugehörigen Zielknoten $V' = \{v_1, \ldots, v_k\}$ zu finden, so dass alle Pfade P_{u_i,v_i} über eine gleiche Kante e führen. Da jede Kante in jedem Schritt nur je ein Paket in jede Richtung transportieren kann, ergibt sich hieraus eine untere Schranke von $k/2$. Wir werden sehen, dass man $k = \sqrt{N}/d$ solche Pfade finden kann.

Man betrachte einen beliebigen Knoten v und alle $N-1$ Pfade $P_{u,v}$, die von allen anderen Knoten zu ihm führen. Für $k \geqslant 1$ bezeichne $S_k(v)$ die Menge aller Kanten, durch die mindestens k dieser Pfade führen. $S_k^*(v)$ bezeichne die Menge aller Endknoten der Kanten in $S_k(v)$. Offensichtlich ist $|S_k^*(v)| \leqslant 2|S_k(v)|$.

Der Beweis wird in mehreren Schritten geführt.

1. Da $N-1$ Pfade zu v hinführen, aber nur d Kanten, müssen über mindestens eine dieser Kanten mindestens $\frac{N-1}{d}$ Pfade führen. Also ist für $k \leqslant \frac{N-1}{d}$ auch $v \in S_k^*(v)$.
2. Es sei von nun an stets $k \leqslant \frac{N-1}{d}$ und daher $v \in S_k^*(v)$.
3. Als nächstes soll gezeigt werden:

$$|V \setminus S_k^*(v)| \leqslant (d-1)(k-1)|S_k^*(v)| \tag{4.1}$$

Wegen der eben gemachten Annahme führt jeder Pfad $P_{u,v}$ von einem Knoten $u \in V \setminus S_k^*(v)$ „nach $S_k^*(v)$ hinein". Für das jeweils erste solche „Hineinführen" über eine Kante $(w, w') \in V \setminus S_k^*(v) \times S_k^*(v)$ gilt:

- Eine solche Kante gehört nicht zu $S_k(v)$, denn sonst wäre ja schon $w \in S_k^*(v)$.
- Es gibt $|S_k^*(v)|$ mögliche w'.
- Zu jedem solchen w' führen maximal $d-1$ Kanten „von außerhalb", denn mindestens eine der mit w' inzidenten Kanten muss ja in $S_k(v)$ liegen.
- Über eine solche Kante (w, w') können höchstens $k-1$ Pfade führen, denn sonst würde diese Kante ja schon zu $S_k(v)$ gehören.

© Th. Worsch 2000-2015

Also kann es „außerhalb“ von $S_k^*(v)$, also in $V \setminus S_k^*(v)$, nur die in Gleichung 4.1 behauptete Anzahl von Knoten geben.

4. Folglich gilt für jedes $k \leqslant (N-1)/d$:

$$\begin{aligned} N &= |V \setminus S_k^*(v)| + |S_k^*(v)| \\ &\leqslant (d-1)(k-1)|S_k^*(v)| + |S_k^*(v)| \\ &\leqslant ((d-1)(k-1)+1) \cdot 2|S_k(v)| \\ &\leqslant 2kd|S_k(v)| \\ \text{und daher} \qquad |S_k(v)| &\geqslant \frac{N}{2kd} . \end{aligned}$$

Summation über alle Knoten ergibt

$$\sum_{v \in V} |S_k(v)| \geqslant \frac{N^2}{2kd} .$$

Da es aber maximal $Nd/2$ Kanten im Graphen gibt, muss mindestens eine Kante in mindestens

$$\frac{N^2/2kd}{Nd/2} = \frac{N}{kd^2}$$

Mengen $S_k(v)$ vorkommen. Wir wählen nun k so, dass diese Anzahl gerade wieder k ist, also $k = \sqrt{N}/d$. (Dieses k ist kleiner gleich $(N-1)/d$.)

5. Es sei nun e eine Kante, die in $k = \sqrt{N}/d$ Mengen $S_k(v_1), \dots, S_k(v_k)$ liegt. D. h. über e führen mindestens $k = \sqrt{N}/d$ Pfade zu v_1, und über e führen mindestens $k = \sqrt{N}/d$ Pfade zu v_2, usw..

   Es sei u_1 einer der k Knoten, für die P_{u_1,v_1} über e führt.

   Nach unserer Wahl von k im vorangegangenen Punkt gibt es zu jedem v_i mindestens k Knoten, für die P_{u_i,v_i} über e führt. Daher können wir induktiv u_i festlegen, indem wir verlangen, dass u_i einer der mindestens $k-(i-1)$ Knoten ungleich $u_1, \dots, u_{i-1}$ sei, für die P_{u_i,v_i} über e führt.

Also gibt es mindestens $k = \sqrt{N}/d$ Pfade $P_{u_1,v_1}, \dots, P_{u_k,v_k}$, die alle über die gleiche Kante e führen. ■

4.9 Man könnte argwöhnen, dass es zumindest einen deterministischen datenunabhängigen Algorithmus gibt, für den nur „sehr sehr wenige“ Permutationen tatsächlich „sehr schlimm“ sind, so dass man in Anwendungen „normalerweise“ gar nicht das konstruierte Problem hat. Leider ist das nicht so.

Man kann zeigen, dass es für jeden deterministischen datenunabhängigen Algorithmus sogar $(\sqrt{N}/d)!$ Permutationen gibt, die mindestens $\sqrt{N}/2d$ Routingschritte nötig machen.

Wir beenden den ersten Abschnitt mit einem Überblick über den Rest dieses Kapitels.

In Abschnitt 4.4 werden wir einen ersten randomisierten Algorithmus für Permutationsrouting kennenlernen, bei dem der Erwartungswert für den Zeit, alle Pakete an ihre Ziele zu transportieren, nur $O(\log N)$ ist. Man vergleiche dies mit der unteren Schranke von $\Omega(\sqrt{N}/\log N)$ für deterministische Algorithmen!

Für die Analyse des randomisierten Algorithmus werden wir Chernoff-Schranken benutzen, die Gegenstand von Abschnitt 4.3 sind. Für die dortigen Beweise werden die Ungleichungen von Markov und Chebyshev benötigt, auf die wir zur Vorbereitung in Abschnitt 4.2 kurz eingehen.

 © Th. Worsch 2000-2015

In Abschnitt 4.5 wird genauer auf die probabilistische Methode eingegangen werden, die in Abschnitt 4.6 bei der Analyse eines zweiten randomisierten Algorithmus für Permutationsrouting benutzt werden wird.

4.2 Markov- und Chebyshev-Ungleichung

4.10 SATZ. (MARKOV-UNGLEICHUNG) *Es sei* Y *eine Zufallsvariable, die nur nichtnegative Werte annehme. Dann gilt für alle* $t, k \in \mathbb{R}_+$*:*

$$\mathbf{Pr}\left[Y \geqslant t\right] \leqslant \frac{\mathbf{E}\left[Y\right]}{t} \quad \textit{bzw.} \quad \mathbf{Pr}\left[Y \geqslant k\mathbf{E}\left[Y\right]\right] \leqslant \frac{1}{k} .$$

4.11 BEWEIS. Man betrachte die Zufallsvariable

$$X = \begin{cases} 0 & \text{falls } Y < t \\ 1 & \text{falls } Y \geqslant t \end{cases}$$

Dann ist $X \leqslant Y$ und $\mathbf{E}[X] \leqslant \mathbf{E}[Y]$. Offensichtlich ist $\mathbf{E}[X] = 0 \cdot \mathbf{Pr}[Y < t] + t \cdot \mathbf{Pr}[Y \geqslant t] = t \cdot \mathbf{Pr}[Y \geqslant t]$. Also ist $t \cdot \mathbf{Pr}[Y \geqslant t] \leqslant \mathbf{E}[Y]$ und $\mathbf{Pr}[Y \geqslant t] \leqslant \mathbf{E}[Y]/t$. ■

Weiß man mehr über die zur Rede stehende Zufallsvariable, zum Beispiel ihre Varianz, dann kann man auch schärfere Abschätzungen angeben:

4.12 SATZ. (CHEBYSHEV-UNGLEICHUNG) *Es sei* X *eine Zufallsvariable mit Erwartungswert* μ_X *und Standardabweichung* σ_X*. Dann gilt für alle* $t \in \mathbb{R}_+$*:*

$$\mathbf{Pr}\left[|X - \mu_X| \geqslant t\sigma_X\right] \leqslant \frac{1}{t^2} .$$

4.13 BEWEIS. Die Zufallsvariable $Y = (X - \mu_X)^2$ hat Erwartungswert $\mu_Y = \sigma_X^2$. Die Markov-Ungleichung ist anwendbar und liefert

$$\mathbf{Pr}\left[Y \geqslant t^2 \mu_Y\right] \leqslant \frac{1}{t^2} .$$

Der Wert auf der linken Seite ist aber

$$\mathbf{Pr}\left[Y \geqslant t^2 \mu_Y\right] = \mathbf{Pr}\left[(X - \mu_X)^2 \geqslant t^2 \sigma_X^2\right] = \mathbf{Pr}\left[|X - \mu_X| \geqslant t\sigma_X\right]$$

■

4.3 Chernoff-Schranken

Die Bezeichnung Chernoff-Schranken geht auf die Arbeit von Chernoff (1952) zurück, der die ersten derartigen Resultate bewies. Heute fasst man den Begriff etwas weiter. Eine kompakte Zusammenfassung entsprechender Ergebnisse stammt von Hagerup und Rüb (1990).

 © Th. Worsch 2000-2015

4.14 Im folgenden seien stets $X_1, \dots, X_n$ unabhängige 0-1-Zufallsvariablen mit $\mathbf{Pr}[X_i = 1] = p_i$ für $1 \leqslant i \leqslant n$. Solche Zufallsvariablen heißen auch *Poisson-Versuche*. Außerdem sei $X = X_1 + \cdots + X_n$ und $\mu = \mathbf{E}[X] = \sum_{i=1}^{n} p_i$. Im Fall, dass alle $p_i = p$ sind, spricht man auch von *Bernoulli-Versuchen*, und X ist binomialverteilt.

In letzterem Fall ist also $\mu = np$,

$$\mathbf{Pr}[X = k] = \binom{n}{k} p^k (1-p)^{n-k} \qquad \text{und} \qquad \mathbf{Pr}[X \geqslant k] = \sum_{j=k}^{n} \binom{n}{j} p^j (1-p)^{n-j} .$$

4.15 Satz. *Mit den Bezeichnungen wie in Punkt 4.14 gilt:*

1. *für* $0 \leqslant \delta$: $\quad \mathbf{Pr}[X \geqslant (1+\delta)\mu] \leqslant \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right)^{\mu}$.

2. *für* $1 > \delta \geqslant 0$: $\quad \mathbf{Pr}[X \leqslant (1-\delta)\mu] \leqslant \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}} \right)^{\mu}$.

In Abbildung 4.1 sind die beiden auf den rechten Seiten auftretenden Abbildungen für den Bereich zwischen 0 und 3 bzw. zwischen 0 und 1 geplottet.

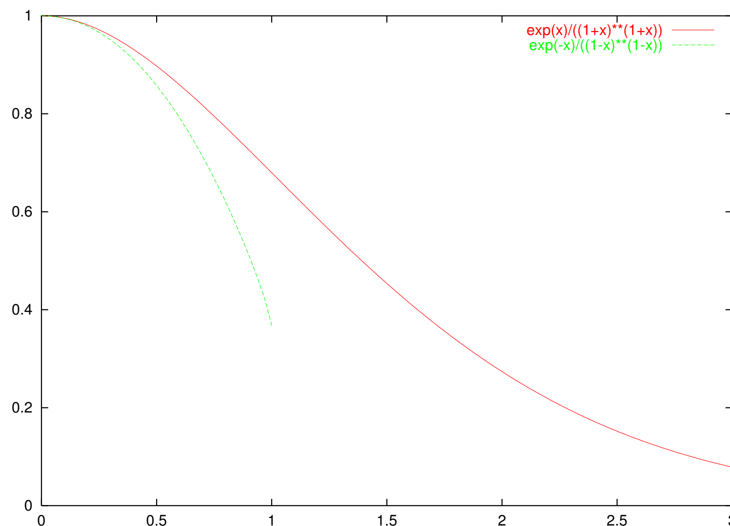


Abbildung 4.1: Die Funktionen $\frac{e^x}{(1+x)^{(1+x)}}$ und $\frac{e^{-x}}{(1-x)^{(1-x)}}$.

4.16 Dem Beweis sei kurz die folgende Überlegung vorangeschickt, die zeigt, dass für $x \geqslant 0$ stets $1 + x \leqslant e^x$ ist: Die Ableitung von $f(x) = e^x - (x+1)$ ist $e^x - 1$, ist also nichtnegativ für $x \geqslant 0$. Folglich nimmt $f(x)$ im Bereich $x \geqslant 0$ für $x = 0$ den minimalen Wert $f(0) = 0$ an. Also ist dort $f(x) \geqslant 0$.

Analog kann man zeigen, dass für $x \geqslant 0$ stets $1 - x \leqslant e^{-x}$ ist.

4.17 Beweis. (von Satz 4.15) Wie die ähnlichen Formeln in Satz 4.15 schon vermuten lassen, können die beiden Ungleichungen ähnlich bewiesen werden.

 © Th. Worsch 2000-2015

1. Mit Hilfe der Markov-Ungleichung erhält man zunächst für jede positive Konstante t:

$$\mathbf{Pr}\left[X \geqslant (1+\delta)\mu\right] = \mathbf{Pr}\left[e^{tX} \geqslant e^{t(1+\delta)\mu}\right] \leqslant \frac{\mathbf{E}\left[e^{tX}\right]}{e^{t(1+\delta)\mu}} .$$

   Mit den X_i sind auch die e^{tX_i} unabhängige Zufallsvariablen. Deshalb kann man den Zähler umformen gemäß:

$$\mathbf{E}\left[e^{tX}\right] = \mathbf{E}\left[e^{t\sum X_i}\right] = \mathbf{E}\left[\prod e^{tX_i}\right] = \prod \mathbf{E}\left[e^{tX_i}\right] .$$

   Weiter ist

$$\mathbf{E}\left[e^{tX_i}\right] = p_i \cdot e^t + (1-p_i)\cdot 1 = 1 + p_i(e^t - 1)$$

   woraus sich mit der Abschätzung $1+x \leqslant e^x$ (siehe Punkt 4.16) ergibt:

$$\mathbf{E}\left[e^{tX_i}\right] \leqslant e^{p_i(e^t-1)}$$

   Damit erhält man:

$$\mathbf{Pr}\left[X \geqslant (1+\delta)\mu\right] \leqslant \frac{\prod e^{p_i(e^t-1)}}{e^{t(1+\delta)\mu}} = \frac{e^{\sum p_i(e^t-1)}}{e^{t(1+\delta)\mu}} = \frac{e^{\mu(e^t-1)}}{e^{t(1+\delta)\mu}} .$$

   Die gewünschte Ungleichung ergibt sich durch Einsetzen von $t = \ln(1+\delta)$. Man beachte, dass dieser Wert tatsächlich positiv ist.

2. Für jede positive Konstante t ist

$$\mathbf{Pr}\left[X \leqslant (1-\delta)\mu\right] = \mathbf{Pr}\left[\mu - X \geqslant \delta\mu\right] = \mathbf{Pr}\left[e^{t(\mu-X)} \geqslant e^{t\delta\mu}\right] \leqslant \frac{\mathbf{E}\left[e^{t(\mu-X)}\right]}{e^{t\delta\mu}} = \frac{\mathbf{E}\left[e^{-tX}\right]}{e^{t(\delta-1)\mu}} .$$

   Ähnlich wie im ersten Fall ist

$$\begin{aligned} \mathbf{E}\left[e^{-tX}\right] &= \prod \mathbf{E}\left[e^{-tX_i}\right] = \prod (p_i e^{-t} + (1-p_i)) = \prod (1 - p_i(1-e^{-t})) \\ &\leqslant \prod e^{-p_i(1-e^{-t})} = e^{\sum -p_i(1-e^{-t})} = e^{-\mu(1-e^{-t})} . \end{aligned}$$

   Die gewünschte Ungleichung ergibt sich Einsetzen von $t = -\ln(1-\delta)$. Man beachte, dass dieser Wert tatsächlich positiv ist.

■

4.18 Bemerkung. Zur Vorbereitung von gelegentlich einfacher handzuhabenden Korollaren zeigen wir zunächst einige technische Ergebnisse. In Satz 4.15 spielt der Ausdruck $F(\mu,\delta) = \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$ für $\delta > -1$ eine Rolle. In diesem Bereich ist $F(\mu,\delta) > 0$.

Dieser Ausruck soll nun nach oben durch etwas gröbere, aber leichter handzuhabende Ausdrücke abgeschätzt werden. Durch Potenzieren mit $1/\mu$ und Logarithmieren ergibt sich der Ausdruck $f(\delta)$, wobei $f(x)$ die Funktion $f(x) = x - (1+x)\ln(1+x)$ ist.

Da Potenzieren mit $1/\mu$ und μ sowie Logarithmieren und Exponentiation für nichtnegative Argumente monotone Funktionen sind, folgt aus einer Abschätzung $f(x) \leqslant k(x)$ auch eine Abschätzung für den interessierenden Ausdruck $F(\mu,\delta) = e^{f(\delta)\mu} \leqslant e^{k(\delta)\mu}$.

Im folgenden werden $f(x)$ und $g(x) = f(x)/x^2$ näher untersucht. In Abbildung 4.2 sind sie und zwei quadratische Polynome geplottet.

 © Th. Worsch 2000-2015

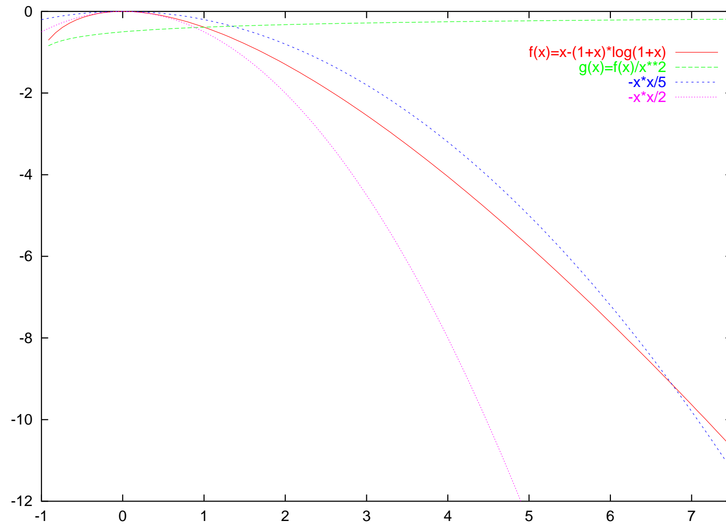


Abbildung 4.2: Die Funktionen $f(x) = x - (1+x)\ln(1+x)$, $g(x) = f(x)/x^2$, $-x^2/5$ und $-x^2/2$ im Vergleich.

4.19 Lemma.

1. Für $-1 < x \leqslant 0$ gilt: $f(x) \leqslant -x^2/2$.
2. Für $0 < x$ gilt: $-x^2/2 \leqslant f(x)$.
3. Die Funktion $g(x) = f(x)/x^2$ ist monoton wachsend.
4. Für $0 < \delta < x$ gilt: $f(\delta) \leqslant g(x)\delta^2$.
5. Für $0 < \delta < 2e - 1$ gilt: $f(\delta) \leqslant -\delta^2/5$.
6. Für $0 < \delta < 1$ gilt: $f(\delta) \leqslant -\delta^2/3$.

4.20 Beweis.

1. Es sei $-1 < x \leqslant 0$. Man betrachte $h(x) = f(x) + x^2/2$. Wegen $\frac{d}{dx} x \ln x = 1 \cdot \ln x + x\frac{1}{x} = 1 + \ln x$ ist $h'(x) = 1 - 1 + \ln(1+x) + x = x + \ln(1+x)$. Daraus folgt $h''(x) = 1 - \frac{1}{1+x}$. Im betrachteten Intervall $-1 < x \leqslant 0$ ist $h''(x) \leqslant h''(0) = 0$. Also ist dort h' fallend, d. h. $h'(x) \geqslant h'(0) = 0$ und daher h steigend, d. h. $h(x) \leqslant h(0) = 0$.
2. Für $x \geqslant 0$ ist analog $h''(x) \geqslant h''(0) = 0$, also $h'(x) \geqslant h'(0) = 0$ und daher $h(x) \geqslant h(0) = 0$.
3. Im folgenden sei stets $x \geqslant 0$ und statt $\ln(1+x)$ schreiben wir kurz $l(x)$. Es ist $g(0) = 0$. Wir zeigen, dass $g'(x) \geqslant 0$ ist. Die Quotientenregel ergibt $\frac{dg}{dx} = \frac{-x^2 l(x) - 2x^2 + 2x(1+x)l(x)}{x^4}$. Dies ist offensichtlich genau dann größer gleich Null, wenn $h(x) = xl(x) + 2l(x) - 2x$ (Zähler durch x) größer gleich Null ist. Es ist $h(0) = 0$. Wir zeigen, dass $h'(x) \geqslant 0$ ist. Ableiten ergibt $h'(x) = -x/(1+x) + l(x)$, also $h'(0) = 0$ und $h''(x) = x/(1+x)^2 \geqslant 0$. Hieraus folgt das Gewünschte.
4. Die Behauptung ist eine einfache Umformulierung der eben gezeigten Monotonieeigenschaft.
5. Einfaches Nachrechnen ergibt

$$g(2e-1) = \frac{2e - 1 - 2e\ln(2e)}{(2e-1)^2} = \frac{2e - 1 - 2e(1 + \ln 2)}{(2e-1)^2} < -\frac{1}{5} \, .$$

   Somit folgt die Behauptung aus dem vorgegangenen Punkt.

 © Th. Worsch 2000-2015

6. Analog zu eben berechnet man $g(1) = f(1) = 1 - 2\ln 2 < -\frac{1}{3}$ und die Behauptung folgt analog wie eben.

■

4.21 Korollar. Mit den Bezeichnungen wie in Punkt 4.14 gilt

$$\text{für } 0 \leqslant \delta \leqslant 2e-1: \quad \mathbf{Pr}\left[X \geqslant (1+\delta)\mu\right] \leqslant \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \leqslant e^{-\delta^2\mu/5}$$

$$\text{für } 0 \leqslant \delta \leqslant 1: \quad \mathbf{Pr}\left[X \geqslant (1+\delta)\mu\right] \leqslant \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \leqslant e^{-\delta^2\mu/3}$$

$$\text{und für } 1 > \delta \geqslant 0: \quad \mathbf{Pr}\left[X \leqslant (1-\delta)\mu\right] \leqslant \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu} \leqslant e^{-\delta^2\mu/2} \leqslant \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} .$$

4.22 Beweis. Die nachzuweisenden Ungleichungen lassen sich auf die aus Lemma 4.19 zurückführen (siehe Bemerkung 4.18). ■

Das folgende Korollar liefert weitere noch einfachere Abschätzungen für Abweichungen nach oben. An ihnen wird auch klar, warum in Lemma 4.21 dem Bereich $0 < \delta < 2e - 1$ besondere Aufmerksamkeit gewidmet wurde.

4.23 Korollar. Es ist

$$\text{für } 0 \leqslant \delta: \quad \mathbf{Pr}\left[X \geqslant (1+\delta)\mu\right] \leqslant \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu}$$

$$\text{für } 2e-1 \leqslant \delta: \quad \mathbf{Pr}\left[X \geqslant (1+\delta)\mu\right] \leqslant 2^{-(1+\delta)\mu} .$$

4.24 Beweis. Nach Satz 4.15 ist

$$\mathbf{Pr}\left[X \geqslant (1+\delta)\mu\right] \leqslant \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \leqslant \left(\frac{e \cdot e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} = \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu} .$$

Für $\delta \geqslant 2e - 1$ ergibt sich:

$$\mathbf{Pr}\left[X \geqslant (1+\delta)\mu\right] \leqslant \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu} \leqslant \left(\frac{e}{1+2e-1}\right)^{(1+\delta)\mu} = \left(\frac{1}{2}\right)^{(1+\delta)\mu} .$$

■

4.4 Erster randomisierter Algorithmus

Wir wollen nun einen ersten randomisierten Algorithmus für das betrachtete Routingproblem beschreiben und analysieren.

Die Vorgehensweise ist sehr einfach:

4.25 Algorithmus.

1. Für jedes Paket b_x in Startknoten x wird unabhängig und gleichverteilt zufällig ein Zwischenknoten z_x gewählt.

 © Th. Worsch 2000-2015

2. Unter Verwendung des Bit-Fixing-Algorithmus wird jedes b_x von x nach z_x transportiert.
3. Unter Verwendung des Bit-Fixing-Algorithmus wird jedes b_x von z_x zu Zielknoten $f(x)$ transportiert.

Kommt es zwischendurch zu Staus, so mögen die Pakete jeweils nach der FIFO-Strategie behandelt werden. Kommen mehrere Pakete gleichzeitig in einem Knoten an und müssen weitergeschickt werden, so mögen sie in einer beliebigen Reihenfolge eingeordnet werden.

Man beachte, dass es vorkommen kann, dass Pakete von mehreren Startknoten zum gleichen Zwischenknoten transportiert werden müssen. Wie sich als Nebenprodukt aus der anschließend durchgeführten Analyse ergeben wird, ist es aber sehr unwahrscheinlich, dass es dadurch zu großen Staus kommt.

4.26 Satz.

1. *Die Wahrscheinlichkeit, dass jedes Paket seinen Zwischenknoten nach spätestens* $7d$ *Schritten erreicht hat, ist mindestens* $1 - 2^{-5d}$.
2. *Die Wahrscheinlichkeit, dass jedes Paket sein Ziel nach spätestens* $14d$ *Schritten erreicht hat, ist mindestens* $1 - \frac{2}{N^5}$.
3. *Für* $d \geqslant 3$ *ist der Erwartungswert für die Laufzeit von Algorithmus 4.25 kleiner oder gleich* $14d + 1$.

Wesentliche Teile des Beweises handeln wir in vorangestellten Lemmata ab. Dazu sei für jeden Startknoten x zufällig ein z_x gewählt und ρ_x bezeichne den Pfad von x nach z_x gemäß dem Bit-Fixing-Algorithmus.

4.27 Lemma. Es sei x beliebig aber fest und $\rho_x = (e_1, e_2, \ldots, e_k)$. Es sei $S_x = \{b_y \mid y \neq x \wedge \rho_y \cap \rho_x \neq \emptyset\}$, wobei die unsaubere Schreibweise $\rho_y \cap \rho_x$ zu interpretieren sei als die Menge der Kanten, die in beiden Pfaden irgendwo vorkommen. Bezeichnet t den tatsächlichen Ankuftszeitpunkt von b_x in z_x, so ist die dann aufgelaufene „Verspätung“ $\ell_x = t - k$ kleiner oder gleich $|S_x|$.

4.28 Beweis.

1. Haben zwei Pfade ρ_x und ρ_y eine oder mehrere Kanten gemeinsam, so gilt: Sobald sich die Pfade das erste Mal wieder getrennt haben, führen sie nicht wieder zusammen.

   Man betrachte die letzte gemeinsame Kante vor der ersten Trennung und den Knoten v, zu dem sie hinführt. Es gibt zwei Fälle:

   - Bei beiden Pfaden folgt noch mindestens eine weitere Kante. Da es sich um verschiedene Kanten handelt, werden verschiedene Bits der Adresse v geändert. Es seien j_1 und $j_2 > j_1$ die Positionen dieser Bits. Also wird auf dem einen Pfad das j_1-te Bit geändert, auf dem anderen aber nicht. Die Pfade ρ_x und ρ_y führen also in die disjunkten Hyperwürfel, die durch eine 0 bzw. 1 als j_1-ten Adressbit gekennzeichnet sind, und verlassen sie nie wieder, da bei beiden nur noch andere Adressbits angepasst werden.
   - Nur ein Pfad führt weiter, der andere endet in v. In diesem Fall kann man analog argumentieren.

2. Wir vereinbaren zunächst folgende Sprechweisen:

 © Th. Worsch 2000-2015

- Ein Paket $b_y \in S_x$ *verlasse* ρ_x, wenn es zum letzten Mal eine Kante von ρ_x benutzt. Achtung: Man lese den letzten Satz noch einmal und beachte die etwas merkwürdige Wortwahl!

  Wegen der Aussage im ersten Punkt ist der Zeitpunkt des „Verlassens“ für jedes Paket b_y eindeutig.
- Das Paket b_x bzw. ein Paket $b_y \in S_x$ habe beim Transport über Kante e_i von ρ_x *Verspätung* ℓ, falls es erst in Schritt $t = i + \ell$ darüber transportiert wird.

  Für b_x ist die so definierte Verspätung beim Transport über Kante e_i wirklich die Zeitdifferenz zwischen dem frühest möglichen Zeitpunkt $t = i$ der Ankunft am Endpunkt von e_i und dem tatsächlichen Ankunftszeitpunkt $t = i + \ell$. Auf die Pakete $b_y \in S_x$ trifft dies nicht zu. Für sie handelt es sich im allgemeinen nur um „irgendeine“ Zahl.

3. Wir zeigen nun, dass jedes Mal, wenn sich die Verspätung von Paket b_x von ℓ auf $\ell + 1$ erhöht, es ein Paket in S_x gibt, das ρ_x mit Verspätung ℓ verlässt. Wegen der Aussage des ersten Punktes kann dies für jedes Paket in S_x nur ein einziges Mal passieren. Also muss tatsächlich $\ell_x \leqslant |S_x|$ sein.

   Man betrachte nun eine Kante e_i, die von Paket b_x zu einem Zeitpunkt t benutzt werden möchte aber nicht benutzt werden kann, weil ein anderes Paket b_y sie benutzt. Die Verspätung von b_x erhöht sich also von $\ell = t - i$ auf $\ell + 1$, und das ist auch der einzige Fall, in dem das passieren kann. Dann ist die „Verspätung“ von b_y bei Benutzung dieser Kante $t - i = \ell$.

   Sei nun t' der letzte Zeitpunkt, zu dem ein Paket aus S_x Verspätung ℓ hat. Es sei b dieses Paket und $e_{j'}$ die Kante, die b benutzen „will“; also ist $t' - j' = \ell$.

   Dann gibt es auch ein Paket in S_x, das ρ_x zu diesem Zeitpunkt t' verlässt: Da b Kante $e_{j'}$ benutzen „will“, wird es selbst oder ein anderes Paket diese Kante auch tatsächlich benutzen. Es sei b' dieses Paket. Es hat offensichtlich Verzögerung $t' - j' = \ell$. Würde b' den Pfad ρ_x *nicht* verlassen, dann gäbe es ein Paket, das Kante $e_{j'+1}$ zum Zeitpunkt $t' + 1$ mit Verzögerung $t' + 1 - (j' + 1) = \ell$ benutzen würde. Dies stünde im Widerspruch zur Wahl von t': es sollte der letzte Zeitpunkt sein, zu dem ein Paket Verspätung ℓ hat. Also verlässt b' Pfad ρ_x zum Zeitpunkt t'.

   Wir schreiben daher nun b' zu, bei Paket b_x die Erhöhung der Verspätung von ℓ auf $\ell + 1$ verursacht zu haben. Da b' den Pfad ρ_x verlässt und nie wieder betritt, wird auf diese Weise keinem Paket aus S_x zweimal eine Verspätungserhöhung zugeschrieben. Also ist $\ell_x \leqslant |S_x|$.

■

4.29 Lemma. Es bezeichne H_{xy} die Zufallsvariable mit $H_{xy} = \begin{cases} 1 & \text{falls } \rho_x \cap \rho_y \neq \emptyset \\ 0 & \text{sonst} \end{cases}$. Dann gilt:

1. Die Gesamtverspätung von b_x beim Eintreffen in z_x ist $\ell_x \leqslant \sum_{y \neq x} H_{xy}$.
2. $\mathbf{E}\left[\sum_{y \neq x} H_{xy}\right] \leqslant d/2$.
3. $\mathbf{Pr}\left[\ell_x \geqslant 6d\right] \leqslant 2^{-6d}$.

4.30 Beweis.

1. Dies ist eine einfache Umformulierung von Lemma 4.27.

 © Th. Worsch 2000-2015

2. Es sei nach wie vor $\rho_x = (e_1, \ldots, e_k)$ mit $k \leqslant d$ irgendein Pfad. Die Zufallsvariable $T(e)$ gebe die Anzahl der Pfade ρ_y mit $y \neq x$ an, die über eine Kante e führen. Dann ist $\sum_{y \neq x} H_{xy} \leqslant \sum_{i=1}^{k} T(e_i)$ und folglich $\mathbf{E}\left[\sum_{y \neq x} H_{xy}\right] \leqslant \sum_{i=1}^{k} \mathbf{E}[T(e_i)]$. Wir zeigen nun noch, dass $\mathbf{E}[T(e_i)] \leqslant 1/2$ ist, so dass die Behauptung aus $k \leqslant d$ folgt.

   Die Kante e_i führe o. B. d. A. von einem Knoten mit Adresse $(x_1 \cdots x_r 0 x_{r+2} \cdots x_d)$ zu Knoten $(x_1 \cdots x_r 1 x_{r+2} \cdots x_d)$. Damit ein Pfad von einem Knoten y nach z_y über e_i führt, muss (da der Bit-Fixing-Algorithmus benutzt wird) y von der Form $y = u_1 \cdots u_r 0 x_{r+2} \cdots x_d$ sein und z_y mit dem Präfix $x_1 \cdots x_r 1$ beginnen. Solche Knoten $y \neq x$ gibt es $2^r - 1$. Da die Zwischenknoten alle zufällig gleichverteilt und unabhängig gewählt werden, ist für jedes y die Wahrscheinlichkeit für das Präfix $x_1 \cdots x_r 1$ in z_y stets $2^{-(r+1)}$.

   Also ist $\mathbf{E}[T(e_i)] = \sum_{y \neq x} \mathbf{Pr}[\rho_y \text{ benutzt } e_i] = \sum_{u_1 \cdots u_r \neq x_1 \cdots x_r} \mathbf{Pr}\left[\rho_{u 0 x_{r+2} \cdots x_d} \text{ benutzt } e_i\right] = (2^r - 1) \cdot 2^{-(r+1)} \leqslant 1/2$.

3. Wegen der ersten beiden Punkte ist $\mathbf{E}[\ell_x] \leqslant d/2$. Folglich gilt wegen des zweiten Teils von Korollar 4.23:

$$\mathbf{Pr}[\ell_x \geqslant 6d] = \mathbf{Pr}[\ell_x \geqslant 12 \cdot d/2] \leqslant \mathbf{Pr}[\ell_x \geqslant (1+11)\mathbf{E}[\ell_x]] \leqslant 2^{-12d/2} = 2^{-6d}$$

■

4.31 Beweis. (von Satz 4.26)

1. Nach Lemma 4.29 ist die Wahrscheinlichkeit, dass ein Paket um mehr als $6d$ Schritte verzögert wird, höchstens 2^{-6d}.

   Insgesamt werden $N = 2^d$ Pakete unabhängig voneinander transportiert. Die Wahrscheinlichkeit, dass wenigstens eines von ihnen um mehr als $6d$ Schritte verzögert wird, ist folglich höchstens $2^d \cdot 2^{-6d} = 2^{-5d}$.

   Da jedes Paket zusätzlich über maximal d Kanten transportiert werden muss, ist auch die Wahrscheinlichkeit, dass wenigstens ein Paket erst nach mehr als $7d$ Schritten am Ziel ist, höchstens 2^{-5d}. Also sind mit Wahrscheinlichkeit $1 - 2^{-5d}$ alle Pakete nach spätestens $7d$ Schritten am Zwischenknoten.

2. Die zweite Phase von Algorithmus 4.25 kann als Umkehrung der ersten Phase aufgefasst werden. Deshalb gilt getrennt hierfür auch die gleiche Analyse. Damit es bei der Nacheinanderausführung beider Phasen nicht zu Effekten kommt, die bei den obigen Abschätzungen nicht berücksichtigt wurden, erweitert man die erste Phase dahingehend, dass jedes Paket nach seiner Ankunft im Zwischenknoten verharrt, bis insgesamt seit Beginn des Routing $7d$ Schritte vergangen sind.

   Folglich ist die Wahrscheinlichkeit, dass alle Pakete nach $\leqslant 14d$ Schritten am Ziel sind, mindestens $(1 - 2^{-5d})(1 - 2^{-5d}) = 1 - 2/N^5 + 1/N^{10} \geqslant 1 - 2/N^5$.

3. Die schlimmste Laufzeit, die überhaupt auftreten kann, ist jedenfalls dadurch nach oben beschränkt, dass die Pakete nacheinander jedes einzeln geroutet werden. Der Zeitbedarf hierfür wäre kleiner gleich $2dN$.

   Für $d \geqslant 3$ ist $N \geqslant 8$, also $14d \leqslant 2dN$. Also gilt dann für den Erwartungswert, dass er kleiner oder gleich $(1 - 2/N^5)14d + (2/N^5) \cdot 2dN = 14d - 28d/N^5 + 4d/N^4 \leqslant 14d + 1$ ist.

■

© Th. Worsch 2000-2015

4.5 Die probabilistische Methode

Die Tragweite der sogenannten *probabilistischen Methode* wurde wohl zuerst von Erdős erkannt, dessen Name seitdem eng mit ihr verbunden ist. Das „Standardbuch" über die probabilistische Methode trägt ebendiesen Titel und stammt von Alon und Spencer (1992).

Es gibt (unter anderen?) die beiden folgenden Varianten.

Die eine fußt auf der Beobachtung, dass jede Zufallsvariable X mindestens einen Wert annimmt, der nicht kleiner als $\mathbf{E}[X]$ ist; und ebenso einen Wert, der nicht größer als $\mathbf{E}[X]$ ist. Also *existieren* Ereignisse, für die X die jeweiligen Werte annimmt.

Die andere Variante basiert auf der folgenden Tatsache. Wird aus einem Universum von Objekten zufällig eines ausgewählt und ist die Wahrscheinlichkeit dafür, dass es eine bestimmte Eigenschaft hat, echt größer als Null, dann muss im Universum ein Objekt *existieren*, das diese Eigenschaft hat. (Andernfalls muss die Wahrscheinlichkeit, ein solches Objekt auszuwählen ja Null sein.)

Für die erste Variante wird in einem späteren Kapitel angewendet werden. Für die zweite Variante werden wir im folgenden Abschnitt ein Beispiel kennenlernen.

4.6 Zweiter randomisierter Algorithmus

Um im folgenden kompakter formulieren zu können sprechen wir statt von einem randomisierten Algorithmus für Permutationsrouting in Hyperwürfeln kürzer von einem *RPH-Algorithmus*. Ein RPH-Algorithmus sei *schnell*, wenn seine erwartete Laufzeit in $O(d)$ liegt.

Algorithmus 4.25 benötigt $\Theta(Nd)$ Zufallsbits und ist schnell, hat also erwartete Laufzeit von $O(d)$. In Satz 4.7 haben wir gesehen, dass Algorithmen, die 0 Zufallsbits benutzen (also deterministische Algorithmen) im schlimmsten Fall Laufzeit $\Omega(\sqrt{N}/d)$ haben.

Es stellt sich daher die Frage, ob es RPH-Algorithmen gibt, die weniger als $\Theta(Nd)$ Zufallsbits benutzen und trotzdem schnell sind. Ziel dieses Abschnittes ist der Nachweis, dass $\Theta(d)$ Zufallsbits notwendig und hinreichend für RPH-Algorithmen sind, um das Problem schnell zu lösen.

Wir zeigen zunächst die Notwendigkeit. Anschließend benutzen wir die probabilistische Methode, um auch die Existenz eines entsprechenden Algorithmus nachzuweisen.

4.32 Satz. *Wenn ein RPH-Algorithmus in Würfeln mit* $N = 2^d$ *Knoten nur* k *Zufallsbits benutzt, dann ist seine erwartete Laufzeit in* $\Omega(2^{-k}\sqrt{N}/d)$.

4.33 Beweis. Ein entsprechender RPH-Algorithmus R kann als Wahrscheinlichkeitsverteilung über 2^k deterministischen Algorithmen aufgefasst werden. Dann gibt es mindestens einen, der mit Wahrscheinlichkeit 2^{-k} ausgewählt wird. Er werde mit A bezeichnet. Es sei x diejenige Eingabe für A, für die A Laufzeit $\Omega(\sqrt{N}/d)$ hat. Man betrachte die Bearbeitung von x durch R. Mit einer Wahrscheinlichkeit von mindestens 2^{-k} wird R wie A arbeiten. Also ist der Erwartungswert für die Laufzeit mindestens $\Omega(2^{-k}\sqrt{N}/d)$. ■

4.34 Korollar. Jeder schnelle RPH-Algorithmus muss $\Omega(d)$ Zufallsbits verwenden.

 © Th. Worsch 2000-2015

4.35 Beweis. Damit für irgendeine positive Konstante c gilt, dass $2^{-k}\sqrt{N}/d \leqslant cd$ ist, muss $2^k \geqslant \sqrt{N}/(cd^2)$ sein, also $k \geqslant \log\sqrt{N} - O(\log d)$ bzw. $k \in \Omega(d)$. ■

Nachdem klar ist, dass ein schneller RPH-Algorithmus $\Omega(d)$ Zufallsbits benötigt, wollen wir nun zeigen, dass es (jedenfalls in einem gewissen Sinne) auch tatsächlich einen solchen Algorithmus gibt.

4.36 Satz. *Für jedes* d *gibt es einen schnellen RPH-Algorithmus, der* $3d$ *Zufallsbits benötigt und erwartete Laufzeit* $22d$ *hat.*

Man beachte, dass hier *nicht* die Existenz eines RPH-Algorithmus für Hyperwürfel aller Größen zugesichert wird.

4.37 Beweis. Im folgenden bezeichne $\mathcal{A} = (A_1, \ldots, A_t)$ eine Liste deterministischer PH-Algorithmen. Jedes $\mathcal{A}$ legt einen randomisierten Algorithmus $R_{\mathcal{A}}$ fest, der zufällig gleichverteilt ein $A_i \in \mathcal{A}$ auswählt.

Wir nennen $\mathcal{A}$ ein *effizientes* N-*Schema*, falls für jede zu routende Permutation auf einem Hyperwürfel mit $N = 2^d$ Knoten gilt: Die erwartete Laufzeit von $R_{\mathcal{A}}$, ist höchstens 22d.

Im folgenden wird gezeigt, dass es für jedes N ein effizientes N-Schema mit $t = N^3$ Algorithmen gibt. Folglich benötigt $R_{\mathcal{A}}$ nur $\log t \in O(\log N) = O(d)$ Zufallsbits (um einen der Algorithmen A_i auszuwählen).

Zum Nachweis der Existenz eines so kleinen effizienten N-Schemas werden wir die probabilistische Methode verwenden. Dazu betrachte man das folgende Zufallsexperiment: Den in Abschnitt 4.4 vorgestellten ersten RPH-Algorithmus kann man als Menge $\mathcal{B} = \{B_1, \ldots, B_{N^N}\}$ von N^N deterministischen PH-Algorithmen auffassen. Hieraus möge zufällig (mit Zurücklegen) eine Liste $\mathcal{A} = (A_1, \ldots, A_{N^3})$ von N^3 Algorithmen $A_i = B_{j_i}$ ausgewählt werden. Wir werden nun zeigen, dass die Wahrscheinlichkeit, dass $\mathcal{A}$ ein effizientes N-Schema ist, echt größer Null ist. Also existiert ein solches.

Es seien die π_i (mit $1 \leqslant i \leqslant N!$) alle $N!$ möglichen zu routenden Permutationen. Ein deterministischer PH-Algorithmus heiße *gut* für ein π_i, wenn diese Permutation in höchstens 14d Schritten durchführt wird, und andernfalls *schlecht*. Aus dem zweiten Teil von Satz 4.26 wissen wir, dass für jedes π_i höchstens ein Bruchteil von $1/N$ aller B_j schlecht für π_i ist.

Es sei zunächst ein beliebiges π_i fixiert. Aus dem eben Gesagten folgt, dass der Erwartungswert für die Anzahl der für π_i schlechten Algorithmen in $\mathcal{A}$ höchstens $N^3/N = N^2$ ist. Es bezeichne X_j die 0-1-Zufallsvariable, die genau dann 1 ist, falls A_j schlecht für π_i ist. Dann ist also $\mu = \mathbf{E}\left[\sum_{j=1}^{N^3} X_j\right] \leqslant N^2$. Die X_j sind unabhängige Zufallsvariablen. Wir wollen nun eine obere Schranke für $\mathbf{Pr}\left[\sum_{j=1}^{N^3} X_j > 4N^2\right]$ beweisen. Dazu sei $c = N^2/\mu \geqslant 1$. Nach einigen Umformungen[1] kann man z. B. die Chernoff-Schranke in der Form aus Korollar 4.23 mit $\delta = 3$

[1] Dank an D. Hoske und S. Walzer für die Korrektur

 © Th. Worsch 2000-2015

benutzen:

$$\begin{aligned}
\mathbf{Pr}\left[\sum_{j=1}^{N^3} X_j > (1+3)N^2\right] &= \mathbf{Pr}\left[\sum_{j=1}^{N^3} X_j > (1+\delta)c\mu\right] = \mathbf{Pr}\left[\sum_{j=1}^{N^3} X_j > (c+c\delta)\mu\right] \\
&\leqslant \mathbf{Pr}\left[\sum_{j=1}^{N^3} X_j > (1+c\delta)\mu\right] \\
&\leqslant \left(\frac{e^{c\delta}}{(1+c\delta)^{1+c\delta}}\right)^{\mu} \leqslant \left(\frac{e^{c\delta}}{(1+c\delta)^{c\delta}}\right)^{\mu} \\
&= \left(\frac{e}{1+c\delta}\right)^{c\delta\mu} \leqslant \left(\frac{e}{1+3}\right)^{c\delta\mu} \\
&= \left(\frac{e}{4}\right)^{3N^2} \leqslant \left(\frac{1}{e}\right)^{N^2}
\end{aligned}$$

Es sei nun E_i das schlechte Ereignis, dass mehr als $4N^2$ Algorithmen in $\mathcal{A}$ schlecht für π_i sind. Dann ist also $\mathbf{Pr}[E_i] < e^{-N^2}$.

Die Wahrscheinlichkeit, dass $\mathcal{A}$ für mindestens ein π_i schlecht ist, ist dann

$$\mathbf{Pr}\left[\bigcup_{i=1}^{N!} E_i\right] \leqslant \sum_{i=1}^{N!} \mathbf{Pr}[E_i] \leqslant N! \cdot e^{-N^2} < 1 .$$

Den Nachweis für die letzte Abschätzung werden wir im Anschluss an diesen Beweis skizzieren. Aus der Abschätzung folgt, dass die Wahrscheinlichkeit, dass von den Algorithmen in $\mathcal{A}$ für jede Permutation höchstens $4N^2$ schlecht sind, positiv ist. Also existiert ein solches $\mathcal{A}$.

Es bleibt nun noch zu zeigen, dass dieses $\mathcal{A}$ sogar ein effizientes N-Schema ist: Dazu sei π_i eine beliebige Permutation. Mit Wahrscheinlichkeit $1-(4N^2/N^3) = 1-4/N$ wird $R_{\mathcal{A}}$ diese Permutation in höchstens $14d$ Schritten durchführen. Andernfalls werden höchstens $2dN$ Schritte benötigt. Der Erwartungswert für die Laufzeit ist daher höchstens

$$(1-\frac{4}{N})14d + \frac{4}{N}2dN \leqslant 22d .$$

■

4.38 Es bleibt noch zu skizzieren, dass für hinreichend große N gilt: $N! \cdot e^{-N^2} < 1$.

Hierzu kann man sich der Stirlingschen Formel bedienen. Sie besagt:

$$N! = \sqrt{2\pi N}\frac{N^N}{e^N}(1+h(N)) \text{ mit } h(N) = \frac{1}{12N} + \frac{1}{288N^2} - \frac{139}{5140N^3} \pm \cdots \in O(\frac{1}{N})$$

Mit anderen Worten ist

$$\lim_{N\to\infty} \frac{N! \cdot e^N}{\sqrt{2\pi N} \cdot N^N} = 1$$

woraus sofort die Behauptung folgt. Genaueres Nachrechnen zeigt, dass schon für $N \geqslant 4$ gilt: $N! \cdot e^{-N^2} < 1$.

4.39 Satz 4.36 behauptet nur die *Existenz* eines – noch dazu nichtuniformen – schnellen RPH-Algorithmus, der nur $O(d)$ Zufallsbits braucht. Sehr viel schwieriger scheint es zu sein, explizit RPH-Algorithmen anzugeben, die möglichst wenige Zufallsbits benötigen. Der beste bislang bekannte immerhin noch $\Theta(d^2)$ Zufallsbits, ist dafür allerdings uniform. Es ist eine nach wie vor ungelöste Aufgabe, diesen Wert zu senken.

© Th. Worsch 2000-2015

Zusammenfassung

1. Beim Routing-Problem in Hyperwürfeln gibt es für jeden deterministischen Algorithmus Permutationen mit *sehr* langen Laufzeiten. Die kann man mit randomisierten Algorithmen im Erwartungswert vermeiden.

2. Mit Hilfe der probabilistischen Methode kann man zeigen, dass ein nichtuniformer RPH-Algorithmus existiert, der größenordnungsmäßig minimale Anzahl von Zufallbits benötigt, aber trotzdem kleine erwartete Laufzeit hat.

Literatur

Alon, Noga und J. Spencer (1992). *The Probabilistic Method*. Wiley Interscience (siehe S. 37).

Borodin, A. und J. E. Hopcroft (1985). „Routing, Merging, and Sorting on Parallel Models of Computation“. In: *Journal of Computer and System Sciences* 30, S. 130–145 (siehe S. 27).

Chernoff, Herman (1952). „A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations“. In: *Annals of Mathematical Statistics* 23, S. 493–507 (siehe S. 29).

Hagerup, T. und C. Rüb (1990). „A Guided Tour of Chernoff Bounds“. In: *Information Processing Letters* 33, S. 305–308 (siehe S. 29).

Kaklamanis, C., D. Krizanc und T. Tsantilas (1990). „Tight Bounds for Oblivious Routing in the Hypercube“. In: *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*. S. 31–36 (siehe S. 27).

Leighton, Frank Thomson (1992). *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. San Mateo, CA 94403: Morgan Kaufmann Publ. ISBN: 1-55860-117-1 (siehe S. 27).

 © Th. Worsch 2000-2015

5 Zwei spieltheoretische Aspekte

In diesem Kapitel wollen wir uns mit dem algorithmischen Problem beschäftigen, sogenannte Und-Oder-Bäume (kurz UOB) auszuwerten. Sie sind ein Spezialfall von Spielbäumen, der selbst aber auch eine Rolle spielt, zum Beispiel in manchen Theorembeweisern. Wir beschränken uns der Bequemlichkeit halber im folgenden auf Bäume mit Verzweigungsgrad 2. Man kann aber analoge Ergebnisse allgemein für Verzweigungsgrad $d \geqslant 2$ beweisen.

5.1 Und-Oder-Bäume und ihre deterministische Auswertung

5.1 Definition Für $k \geqslant 1$ sei T_k der wie folgt rekursiv definierte vollständige binäre Baum der Höhe $2k$, dessen innere Knoten abwechselnd mit "$\wedge$" und "$\vee$" markiert sind.

- Die Wurzel von T_1 ist ein $\wedge$-Knoten und hat zwei $\vee$-Knoten als Nachfolger. Jeder dieser Knoten hat zwei Blätter als Nachfolger.
- Für $k \geqslant 2$ ergibt sich T_k aus T_1, indem man dessen Blätter durch Kopien von T_{k-1} ersetzt. $\diamond$

Wie man leicht sieht, könnte man in obiger Definition den Rekursionschritt auch völlig äquivalent so formulieren:

- Für $k \geqslant 2$ und $1 \leqslant l < k$ ergibt sich T_k aus T_l, indem man dessen Blätter durch Kopien von T_{k-l} ersetzt.

Im folgenden bezeichne n stets die Anzahl der Blätter eines UOB. T_k besitzt also $n = 4^k$ Blätter, die im folgenden mit $x_1, \dots, x_{4^k}$ bezeichnet werden.

5.2 Durch die Festlegung von booleschen Werten an allen Blättern eines UOB wird auf naheliegende Weise auch für alle inneren Knoten und damit auch für die Wurzel des Baumes ein Wert festgelegt.

In den beiden ersten Abschnitten dieses Kapitels wollen wir uns mit deterministischen und einem randomisierten Algorithmus zur Bestimmung der Wurzelwerte von UOB beschäftigen. Dabei wollen wir uns insbesondere dafür interessieren, wieviele Blätter der Algorithmus besucht, um den Wurzelwert zu bestimmen.

5.3 Offensichtlich kann durch den Besuch aller $n = 4^k$ Blätter und die Berechnung der Werte aller inneren Knoten „bottom up" den der Wurzel bestimmen.

Zunächst stellt sich die Frage, ob deterministische Algorithmen auch geschickter vorgehen können. Die Antwort ist nein:

5.4 Satz. *Für jedes $k \geqslant 1$ und jeden deterministischen Algorithmus A zur Auswertung von UOB gilt: Es gibt eine Folge $x_1, \dots, x_{4^k}$ von Bits, so dass A bei der Auswertung von T_k mit den x_i als Blattwerten* alle $n = 4^k$ *Blätter besucht. Dabei ist der Wert der Wurzel gleich dem des zuletzt besuchten Blattes und es kann also sowohl erzwungen werden, dass dieser gleich 0 ist, als auch, dass er gleich 1 ist.*

5.5 Beweis. Durch Induktion:

$k = 1$: Es ist klar, dass A mindestens ein Blatt besuchen muss. O. B. d. A. sei dies x_1. Wir setzen $x_1 = 0$. Damit ist weder der Wert des übergeordneten $\vee$-Knotens noch der der Wurzel bereits festgelegt und A muss ein weiteres Blatt besuchen. Wieder gibt es o. B. d. A. zwei Möglichkeiten:

1. Das als zweites besuchte Blatt ist x_2. Wir setzen $x_2 = 1$. Damit ist nur der Wert des übergeordneten $\vee$-Knotens klar aber noch nicht der der Wurzel und A muss ein weiteres Blatt besuchen. O. B. d. A. sei dies x_3. Wir setzen $x_3 = 0$. Damit muss A auch noch x_4 besuchen, denn dessen Wert ist der der Wurzel.
2. Das als zweites besuchte Blatt ist x_3. Wir setzen $x_3 = 0$. Damit ist weder der Wert des übergeordneten $\vee$-Knotens noch der der Wurzel bereits festgelegt und A muss ein weiteres Blatt besuchen. O. B. d. A. sei dies x_2. Wir setzen $x_2 = 1$. Damit muss A auch noch x_4 besuchen, denn dessen Wert ist der der Wurzel.

$k - 1 \rightsquigarrow k$: Wir fassen T_k auf als einen T_1-Baum, dessen Blätter durch T_{k-1}-Bäume ersetzt sind. Wir bezeichnen die „Blätter“ von T_1 mit $y_1, \dots, y_4$. Analog zur überlegung für den Induktionsanfang ist klar, dass A mindestens einen der Werte y_i bestimmen muss. Mehr noch, man kann durch geschickte Wahl der y_i in Abhängigkeit von der Reihenfolge, in der A sie berechnet, erzwingen, dass A sogar *alle* Werte y_1, y_2, y_3 und y_4 ermitteln muss. Nach Induktionsvoraussetzung gibt es für jeden der T_{k-1}-Bäume eine Belegung der Blattwerte, die das gewünschte y_i liefert und gleichzeitig erzwingt, dass A zu dessen Berechnung jeweils alle darunter liegenden Blätter besuchen muss.

Also muss A in diesem Fall alle Blätter überhaupt besuchen.

■

5.2 Analyse eines randomisierten Algorithmus für die Auswertung von UOB

5.6 Algorithmus.

proc *AndNodeEval*(T)
if *IsLeaf*(T) **then return** *value*(T) **fi**
⟨*andernfalls:*⟩
T' ← ⟨*zufällig gewählter Unterbaum von* T⟩
r ← *OrNodeEval*(T')
if $r = 0$ **then**
  return 0
else
  T'' ← ⟨*der andere Unterbaum von* T⟩
  return *OrNodeEval*(T'')
fi

proc *OrNodeEval*(T)

 © Th. Worsch 2000-2015

$T' \leftarrow \langle$*zufällig gewählter Unterbaum von* $\mathsf{T}\rangle$
$r \leftarrow$ *AndNodeEval*(T')
if $r = 1$ **then**
  return 1
else
  $T'' \leftarrow \langle$*der andere Unterbaum von* $\mathsf{T}\rangle$
  return *AndNodeEval*(T'')
fi
AndNodeEval$(root)$

5.7 SATZ. *Der Erwartungswert für die Anzahl der von Algorithmus 5.6 besuchten Blätter für einen* T_k*-Baum ist für jede Folge* $x_1, \dots, x_{4^k}$ *von Blattwerten höchstens* $3^k = n^{\log_4 3} \approx n^{0.792\dots}$.

5.8 BEWEIS. Durch Induktion.

$k = 1$**:** Diesen Fall erledigt man durch systematisches überprüfen aller 16 möglichen Kombinationen für die $x_1, \dots, x_4$. Beispielhaft betrachten wir den Fall 0100:

1. Falls zuerst der linke Teilbaum ausgewertet wird: Mit gleicher Wahrscheinlichkeit wird erst und nur die 1 oder erst die 0 und danach die 1 besucht. Anschließend werden im rechten Teilbaum beide Blätter besucht. Erwartungswert: 7/2.
2. Falls zuerst der rechte Teilbaum untersucht wird: Nach dem Besuch beider Blätter ist klar, dass der T_1-Baum den Wert 0 liefert. Erwartungswert: 2.

Da beide Fälle gleich wahrscheinlich sind, ergibt sich insgesamt $1/2 \cdot 7/2 + 1/2 \cdot 2 = 11/4 < 3$.

$k - 1 \rightsquigarrow k$**:** Wir betrachten zunächst nicht einen ganzen T_k-Baum, sondern einen $\vee$-Knoten, an dem zwei T_{k-1}-Bäume „hängen". Es gibt zwei Fälle:

O1. Der $\vee$-Knoten wird eine 1 liefern: Dann muss mindestens einer der T_{k-1}-Bäume dies auch tun. Da gleichwahrscheinlich jeder der beiden zuerst untersucht wird, wird mit einer Wahrscheinlichkeit $p \geqslant 1/2$ als erstes ein (und nur ein) Unterbaum untersucht, der eine 1 liefert. Mit Wahrscheinlichkeit $1 - p \leqslant 1/2$ werden beide Unterbäume untersucht. Der Erwartungswert ist also höchstens $p \cdot 3^{k-1} + (1 - p) \cdot 2 \cdot 3^{k-1} = (2 - p) \cdot 3^{k-1} \leqslant 3/2 \cdot 3^{k-1}$.

O2. Der $\vee$-Knoten wird eine 0 liefern: Dann müssen beide T_{k-1}-Bäume dies auch tun. Mit der Induktionsvoraussetzung ergibt sich, dass der Erwartungswert für die Anzahl der in diesem Fall besuchten Blätter höchstens $2 \cdot 3^{k-1}$ ist.

Betrachten wir nun die Wurzel des T_k-Baumes, an der zwei der eben untersuchten Bäume hängen. Es gibt zwei Fälle:

U1. Der $\wedge$-Knoten wird eine 0 liefern: Dann muss mindestens einer der Unterbäume dies auch tun. Da gleichwahrscheinlich jeder der beiden zuerst untersucht wird, wird mit einer Wahrscheinlichkeit $p \geqslant 1/2$ als erstes ein (und nur ein) Unterbaum untersucht, der eine 0 liefert. Mit Wahrscheinlichkeit $1 - p \leqslant 1/2$ werden beide Unterbäume untersucht. Gemäß der überlegungen in OO1. und OO2. ist der Erwartungswert folglich höchstens $p \cdot 2 \cdot 3^{k-1} + (1 - p) \cdot (3/2 \cdot 3^{k-1} + 2 \cdot 3^{k-1}) = 7/2 \cdot 3^{k-1} - p \cdot 3/2 \cdot 3^{k-1} \leqslant 11/4 \cdot 3^{k-1} \leqslant 3^k$.

U2. Der $\wedge$-Knoten wird eine 1 liefern: Dann müssen beide Unterbäume dies auch tun. Nach Fall OO1. ist daher der Erwartungswert für die Anzahl besuchter Blätter $2 \cdot 3/2 \cdot 3^{k-1} \leqslant 3^k$.

 © Th. Worsch 2000-2015

5.3 Zwei-Personen-Nullsummen-Spiele

In diesem Abschnitt sind ein wenig Notation und Ergebnisse zu einem Thema aus der Spieltheorie zusammen gestellt.

5.9 Im allgemeinen hat man es mit $n \geqslant 2$ *Spielern* zu tun. Jeder Spieler i hat eine (endliche) Menge S_i sogenannter *reiner Strategien* s^i_j zur Auswahl. Für jeden Spieler i gibt es eine Funktion $u_i : S_1 \times \cdots \times S_n \to \mathbb{R}$, die für jede Kombination von Strategien angibt, welchen *Nutzen* oder *Gewinn* Spieler i hat, wenn die Spieler sich für eine bestimmte Kombination von Strategien entscheiden.

5.10 Bei *Zwei-Personen-Nullsummen-Spielen* gibt es $n = 2$ Spieler und für die Nutzenfunktionen gilt: $u_1 = -u_2$. Es genügt also zum Beispiel u_1 anzugeben; das kann man dann in Form einer Matrix $\mathbf{M}$ mit $|S_1|$ Zeilen und $|S_2|$ Spalten tun, bei der Eintrag M_{ij} gerade $u_1(s^1_i, s^2_j)$ ist.

Deshalb spricht man dann auch manchmal vom *Zeilenspieler* und vom *Spaltenspieler*. Identifiziert man die Wahl einer reinen Strategie i mit dem Einheitsvektor $\mathbf{e}_i$ (jeweils passender Länge und in Spaltenform), dann ist $u_1(s^1_i, s^2_j) = \mathbf{e}_i^\top \mathbf{M} \mathbf{e}_j$.

5.11 Eine *gemischte Strategie* ist eine Wahrscheinlichkeitsverteilung $\mathbf{p}$ auf der Menge der reinen Strategien eines Spielers.

Sind $\mathbf{p}$ und $\mathbf{q}$ gemischte Strategien für Zeilen- und Spaltenspieler, dann ist $\mathbf{p}^\top \mathbf{M} \mathbf{q}$ der zu erwartende Gewinn für den Zeilenspieler.

5.12 Satz. (Neumann 1928) *Für Zwei-Personen-Nullsummen-Spiele mit Matrix* $\mathbf{M}$ *gilt:*

$$\max_{\mathbf{p}} \min_{\mathbf{q}} \mathbf{p}^\top \mathbf{M} \mathbf{q} = \min_{\mathbf{q}} \max_{\mathbf{p}} \mathbf{p}^\top \mathbf{M} \mathbf{q}$$

Wir werden diesen Satz hier nicht beweisen. Man kennt verschiedene Möglichkeiten, es zu tun. Zum Beispiel kann man Verteilungen $\mathbf{p}^*$ und $\mathbf{q}^*$, für die der Wert aus von Neumanns Satz angenommen wird, nach Brouwers Fixpunktsatz als Fixpunkt einer geeigneten Abbildung erhalten.

5.13 Korollar. (Loomis 1946) Für Zwei-Personen-Nullsummen-Spiele mit Matrix $\mathbf{M}$ gilt:

$$\max_{\mathbf{p}} \min_{j} \mathbf{p}^\top \mathbf{M} \mathbf{e}_j = \min_{\mathbf{q}} \max_{i} \mathbf{e}_i^\top \mathbf{M} \mathbf{q}$$

5.14 Beweis. Es genügt zu zeigen, dass für jedes $\mathbf{p}$ gilt: $\min_{\mathbf{q}} \mathbf{p}^\top \mathbf{M} \mathbf{q} = \min_j \mathbf{p}^\top \mathbf{M} \mathbf{e}_j$ und analog für die rechten Seiten der beiden Gleichungen aus Satz 5.12 und Korollar 5.13.

Für beliebiges $\mathbf{p}$ ist $\mathbf{p}^\top \mathbf{M}$ ein Zeilenvektor $\mathbf{v}^\top$. Es sei j eine Stelle in $\mathbf{v}$, an der der kleinste aller in $\mathbf{v}$ vorkommenden Werte steht. Dann ist offensichtlich $\mathbf{v}^\top \mathbf{e}_j$ der kleinste überhaupt mögliche Wert, der für ein $\mathbf{v}^\top \mathbf{q}$ auftreten kann. ∎

Aus Korollar 5.13 ergibt sich offensichtlich die folgende Aussage, die wir im anschließenden Abschnitt ausnutzen werden.

5.15 Korollar. Für alle Verteilungen $\mathbf{p}$ und $\mathbf{q}$ gilt:

$$\min_{j} \mathbf{p}^\top \mathbf{M} \mathbf{e}_j \leqslant \max_{i} \mathbf{e}_i^\top \mathbf{M} \mathbf{q}$$

 © Th. Worsch 2000-2015

5.4 Untere Schranken für randomisierte Algorithmen

Im letzten Abschnitt dieses Kapitels soll eine Technik vorgestellt werden, um untere Schranken für den Ressourcenverbrauch randomisierter Algorithmen nachzuweisen. Tatsächlich handelt es sich wohl um die derzeit einzige solche Methode.

5.16 Stellen Sie sich nun vor, dass es zwei Spieler gibt:

- Spaltenspieler ist jemand der als verschiedene Strategien deterministische Algorithmen A zur Auswahl hat.
- Zeilenspieler ist ein böser Widersacher, der als verschiedene Strategien Eingaben I zur Auswahl hat.

Der Gewinn des Widersachers ist jeweils $C(I, A)$. Das sei zum Beispiel die Laufzeit von Algorithmus A für Eingabe I (oder der Verbrauch irgendeiner anderen Ressource).

Der Widersacher versucht, $C(I, A)$ zu maximieren, der Algorithmenentwerfer versucht, $C(I, A)$ zu minimieren.

- Eine gemischte Strategie des Widersachers ist eine Wahrscheinlichkeitsverteilung auf der Menge der Eingaben.
- Eine gemischte Strategie des Algorithmenentwerfers ist ein randomisierter Algorithmus.

Stellt man sich nun noch vor, dass $\mathbf{M}$ die Werte $C(I, A)$ enthält, dann ist klar:

5.17 Satz. (Minimax-Methode von Yao) *Es sei P ein Problem für eine endliche Menge $\mathcal{I}$ von Eingaben gleicher Größe n und $\mathcal{A}$ eine endliche Menge von Algorithmen für dieses Problem. Für $I \in \mathcal{I}$ und $A \in \mathcal{A}$ bezeichne $C(I, A)$ den Ressourcenverbrauch, z. B. die Laufzeit, von Algorithmus A für Eingabe I.*

Weiter bezeichne $\mathbf{p}$ bzw. $\mathbf{q}$ eine Wahrscheinlichkeitsverteilung auf $\mathcal{I}$ bzw. $\mathcal{A}$. Mit I_p bzw. A_q werde ein gemäß der Verteilung $\mathbf{p}$ bzw. $\mathbf{q}$ aus $\mathcal{I}$ bzw. $\mathcal{A}$ gewählte Eingabe bzw. Algorithmus bezeichnet.

Dann gilt für alle $\mathbf{p}$ und $\mathbf{q}$:

$$\min_{A \in \mathcal{A}} \mathbf{E}\left[C(I_p, A)\right] \leqslant \max_{I \in \mathcal{I}} \mathbf{E}\left[C(I, A_q)\right]$$

5.18 Einige Erläuterungen erscheinen angebracht:

- Der Erwartungswert auf der linken Seite ergibt sich durch die zufällige Wahl von I_p gemäß Verteilung $\mathbf{p}$. Für jeden deterministischen Algorithmus A handelt es sich dabei also um die „erwartete Laufzeit“ von A für gewisse Eingabeverteilungen. Das Minimum der Erwartungswerte, also der Erwartungswert für den „besten“ Algorithmus ist in der Ungleichung von Bedeutung.

- Der Erwartungswert auf der rechten Seite ergibt sich durch die zufällige Wahl von A_q gemäß $\mathbf{q}$. Für jede Eingabe I handelt es sich dabei also um die „erwartete Laufzeit“ gewisser deterministischer Algorithmen für I.

- Wir erinnern an Punkt 1.1. Jeder (randomisierte) Las-Vegas-Algorithmus kann als eine Menge deterministischer Algorithmen aufgefasst werden, aus denen nach einer gewissen Wahrscheinlichkeitsverteilung bei jeder Ausführung einer ausgewählt wird. Das Maximum über verschiedene Eingaben des Erwartungswertes auf der rechten Seite ist also die interessierende Größe.

 © Th. Worsch 2000-2015

- Mit anderen Worten: $\min_{A \in \mathcal{A}} \mathbf{E}[C(I_p, A)]$ ist eine untere Schranke für Laufzeit des randomisierten Algorithmus (für gewisse Eingaben).

5.19 BEMERKUNG. Einen Satz analog zu 5.17 kann man auch für Monte-Carlo-Algorithmen beweisen. Hierauf gehen wir nicht weiter ein.

Wir wollen nun die Minimax-Methode auf das Problem der Auswertung von UOB anwenden.

5.20 Als erstes beobachte man, dass wegen

$$\overline{\overline{(x_1 \vee x_2) \wedge (x_3 \vee x_4)}} = \overline{\overline{(x_1 \vee x_2)} \vee \overline{(x_3 \vee x_4)}} = (x_1 \overline{\vee} x_2) \overline{\vee} (x_3 \overline{\vee} x_4)$$

jeder UOB äquivalent auch als Baum dargestellt werden kann, dessen innere Knoten *alle* die NOR-Funktion $\overline{\vee}$ berechnen.

5.21 Ein $\overline{\vee}$-Gatter liefert genau dann eine 1, wenn an beiden Eingängen eine 0 vorliegt.

Die Zahl $p = \frac{3-\sqrt{5}}{2}$ hat die Eigenschaft $(1-p)^2 = p$ (wie man durch einfaches Nachrechnen sieht). Wenn an jedem Eingang eines $\overline{\vee}$-Gatters unabhängig mit Wahrscheinlichkeit p eine 1 vorliegt, ist daher mit gleicher Wahrscheinlichkeit p auch die Ausgabe eine 1.

Als letzten vorbereitenden Schritt benötigen wir noch die folgende Tatsache.

5.22 SATZ. *Es sei T ein vollständiger balancierter Baum aus $\overline{\vee}$-Knoten, dessen Blätter alle unabhängig voneinander mit einer Wahrscheinlichkeit q den Wert 1 haben. Es sei $W(T)$ das Minimum (genommen über alle deterministischen Algorithmen) der erwarteten Anzahl von Schritten zur Auswertung von T.*

Dann gibt es auch einen Algorithmus A, der eine erwartete Anzahl von nur $W(T)$ Schritten macht und außerdem die folgende Eigenschaft hat: Besucht A ein Blatt v', das zu einem Teilbaum T' gehört und später ein Blatt u, das nicht *zu T' gehört, dann gilt für alle Blätter a'' von T', die A überhaupt besucht: A besucht a'' vor u.*

Damit können wir nun beweisen:

5.23 SATZ. *Die erwartete Anzahl der Blätter, die ein randomisierter Algorithmus zur Auswertung von UOB mit n Blättern besucht, ist mindestens $n^{\log_2((1+\sqrt{5})/2)} = n^{0.694\ldots}$.*

5.24 BEWEIS. Wir betrachten nun einen Algorithmus wie in Satz 5.22 und die Auswertung von $\overline{\vee}$-Bäumen, deren Blätter unabhängig voneinander mit Wahrscheinlichkeit $p = \frac{3-\sqrt{5}}{2}$ auf 1 gesetzt sind. In Abhängigkeit von der Höhe h sei $W(h)$ die erwartete Anzahl besuchter Blätter.

Offensichtlich ist

$$\begin{aligned} W(h) &= W(h-1) + (1-p)W(h-1) = (2-p)W(h-1) \\ \text{also} \quad W(h) &= (2-p)^{h-1}W(1) = (2-p)^h \end{aligned}$$

Einsetzen von $h = \log_2 n$ und p ergibt

$$W(T) = W(\log_2 n) = (2-p)^{\log_2 n} = 2^{(\log_2(2-p))(\log_2 n)} = n^{\log_2(2-p)} = n^{0.694\ldots}$$

■

5.25 Durch eine genauere (und schwierigere) Analyse kann man sich davon überzeugen, dass sogar die obere Schranke von $n^{\log_4 3} \approx n^{0.792\ldots}$ aus Satz 5.7 gleichzeitig auch untere Schranke ist. Algorithmus 5.6 ist also optimal.

 © Th. Worsch 2000-2015

Zum Abschluss dieses Kapitels wollen wir noch auf einen anderen Aspekt aufmerksam machen, der sich hinter Satz 5.7 verbirgt.

5.26 Da der Erwartungswert für die Anzahl besuchter Blätter $n^{0.792\cdots}$ ist, muss es mindestens eine Berechnung geben, während der höchstens so viele Blätter besucht werden. (Würden stets mehr Blätter besucht, könnte der Erwartungswert nicht so klein sein.)

Oder anders formuliert: Mit einer gewissen Wahrscheinlichkeit echt größer 0 findet der randomisierte Algorithmus eine Teilmenge von höchstens $n^{0.792\cdots}$ Blättern, aus deren Werten bereits der der Wurzel folgt.

Also existiert, und zwar *für jede Eingabe* (i. e. Verteilung von Bits auf alle Blätter), eine solche „kleine" Teilmenge von Blättern, deren Kenntnis für die Bestimmung des Wertes an der Wurzel ausreicht.

Andererseits haben wir in Satz 5.4 gesehen, dass jeder deterministische Algorithmus für manche Eingaben alle Blätter besuchen muss. Es ist also manchmal „sehr schwierig", deterministisch eine solche kleine Teilmenge zu finden.

Zusammenfassung

1. Bei der Auswertung von Und-Oder-Bäumen kann man randomisiert weniger Blattbesuche erwarten, als jeder deterministische Algorithmus für manche Bäume durchführen *muss*.
2. Die Minimax-Methode von Yao liefert eine Möglichkeit, untere Schranken für die erwartete Laufzeit randomisierter Algorithmen herzuleiten.

Literatur

Loomis, L. H. (1946). „On a theorem of von Neumann". In: *Proceedings of the National Academy of Sciences of the USA* 32, S. 213–215 (siehe S. 44).

Neumann, John von (1928). „Zur Theorie der Gesellschaftsspiele". In: *Mathematische Annalen* 100, S. 295–320 (siehe S. 44).

 © Th. Worsch 2000-2015

6 Graph-Algorithmen

6.1 DEFINITION Ein *Multigraph* ist eine Struktur (V, E, ν), die aus einer Menge V von Knoten und einer Menge E von Kanten besteht und einer Funktion ν, die jeder Kante die Menge ihrer zwei (nicht notwendig verschiedenen) Endpunkte zuweist.

Bei Multigraphen können zwei Knoten also durch mehrere, wohl unterschiedene Kanten miteinander verbunden sein. Man spricht auch von *Mehrfachkanten*. ◇

In diesem Kapitel sind alle Graphen Multigraphen ohne Schlingen mit *ungerichteten* Kanten.

6.2 Im Folgenden bezeichnen wir stets mit n die Anzahl der Knoten und mit m die Anzahl der Kanten eines Multigraphen. Im Gegensatz zu „normalen" Graphen ist also m im allgemeinen *nicht* durch $O(n^2)$ nach oben beschränkt.

6.3 Multigraphen kann man mit Datenstrukturen repräsentieren, die leichte Erweiterungen von Adjazenzmatrizen bzw. Adjazenzlisten sind, wie man sie für normale Graphen kennt.

In der Adjazenzmatrix A speichert der Eintrag $A[x, y]$, wie viele Kanten es zwischen x und y gibt. Analog kann man bei Adjazenzlisten vorgehen. Weil es weiter unten von Vorteil ist, soll zusätzlich in einer Variablen n die Anzahl Knoten, in einer Variablen m die Gesamtzahl Kanten und in einem Vektor M für jeden Knoten die Zahl der von ihm ausgehenden Kanten gespeichert sein. Für alle x gelte also $M[x] = \sum_y A[x, y]$ und es sei $\sum_x M[x] = 2m$.

Zeilen und Spalten seien jeweils von 1 bis n durchnummeriert.

6.1 Minimale Schnitte

In diesem Abschnitt beschäftigen wir uns mit dem Problem, minimale Schnitte in Multigraphen zu berechnen.

6.4 DEFINITION Ein *Schnitt* in einem Multigraphen $G(V, E)$ ist durch eine Partitionierung $V = C \cup \bar{C}$ der Knotenmenge in zwei diskunkte Teilmengen gegeben. Die *Größe* eines Schnittes ist die Anzahl der Kanten $\{x, y\}$ mit $x \in C$ und $y \in \bar{C}$.

Ein minimaler Schnitt ist ein Schnitt mininaler Größe. ◇

6.5 LEMMA. Wenn in einem Multigraphen die minimalen Schnitte Größe k haben, dann ist auch der Grad jedes Knotens mindestens k. Für die Anzahl der Kanten gilt folglich: $m \geqslant nk/2$.

6.6 BEWEIS. Andernfalls würde die Partitionierung, die in C nur einen Knoten x mit Grad kleiner als k enthält, und in $\bar{C} = V \setminus \{x\}$ alle anderen Knoten, einen Schnitt mit Größe kleiner als k liefern. ■

6.7 DEFINITION Es sei G ein Multigraph und x und y zwei Knoten von G, die durch mindestens eine Kante e miteinander verbunden sind. Der durch die *Kontraktion* von e entstehende Multigraph G/e wie folgt definiert:

- Seine Knotenmenge ist $V' = (V \setminus \{x, y\}) \cup \{z\}$, wobei z ein neuer Knoten ist.
- Seine Kantenmenge ergibt sich aus E, indem
  - alle Kanten zwischen x und y entfernt werden,
  - jede Kante, die in G einen Knoten $v \in V \setminus \{x, y\}$ mit einem Knoten aus $\{x, y\}$ verband, durch eine Kante zwischen v und z ersetzt wird und
  - alle anderen Kanten von G übernommen werden.

Falls man nacheinander alle Kanten einer Kantenteilmenge $X \subset E$ kontrahiert, so ist die Reihenfolge, in der die Kanten kontrahiert für das Ergebnis irrelevant. Daher schreibt man in diesem Fall dann einfach G/X für den kontrahierten Graphen. ◇

Bei der Kontraktion einer Kante verringert sich die Knotenzahl um genau 1.

6.8 Durch die Kontraktion einer Kante können (zusätzliche) Mehrfachkanten entstehen. Abbildung 6.1 zeigt ein einfaches Beispiel.

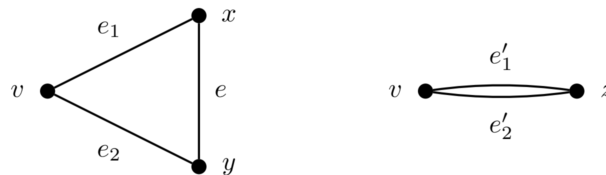


Abbildung 6.1: Durch Kontraktion der Kante e entstehen Mehrfachkanten.

6.9 Lemma. Ist G ein Multigraph mit einer Kante e, so sind die minimalen Schnitte von G/e mindestens so groß wie die von G.

6.10 Beweis. Man betrachte einen minimalen Schnitt $(K, \bar{K})$ von G/e; seine Größe sei k. O. B. d. A. seien die Endknoten x und y von e in K. Indem man diese beiden Knoten „aus der Kontraktion auspackt" und in der gleichen Teilmenge belässt, erhält man einen Schnitt von G, dessen Größe ebenfalls k ist. Also haben die minimalen Schnitte von G *höchstens* Größe k. ■

6.11 Algorithmus. (Kontraktion) Es sei A die Adjazenzmatrix eines Multigraphen G und e eine Kante zwischen x und $y > x$. Die Datenstrukturen für G/e lassen sich wie folgt berechnen:

proc graph ← Kontraktion(**graph** G, **edge** e)
  ⟨*Idee: benutze Zeile/Spalte x für den kontrahierten Knoten*⟩
  ⟨   *und Zeile/Spalte y für das, was bislang in Zeile n stand*⟩
  ⟨*Aktualisierung der Kantenzahlen*⟩
  $m \leftarrow m - A[x, y]$
  $M[x] \leftarrow M[x] + M[y] - 2 \cdot A[x, y]$
  $M[y] \leftarrow M[n]$
  ⟨*Aktualisierung von Zeile/Spalte x:*⟩
  $A[x, \cdot] \leftarrow A[x, \cdot] + A[y, \cdot]$
  $A[\cdot, x] \leftarrow A[\cdot, x] + A[\cdot, y]$
  $A[x, x] \leftarrow 0$
  ⟨*Aktualisierung von Zeile/Spalte y:* ⟩
  $A[y, \cdot] \leftarrow A[n, \cdot]$

© Th. Worsch 2000-2015

$A[\cdot, y] \leftarrow A[\cdot, n]$
⟨*die bisherige Zeile* n *ist nun bedeutungslos*⟩
$n \leftarrow n - 1$
return ⟨*Graph, der zu den neuen Datenstrukturen gehört*⟩

Dieser Algorithmus benötigt offensichtlich Laufzeit $O(n)$.

6.12 Algorithmus. (Iterierte Kontraktion)
⟨*Eingabe: ein Multigraph* $G(V, E)$⟩
⟨*Ausgabe: ein Schnitt* C⟩
$H \leftarrow G$
while (H hat mehr als 2 Knoten und mindestens 1 Kante) **do**
  $e \leftarrow$⟨*zufällig gleichverteilt gewählte Kante von* H⟩
  $H \leftarrow$ Kontraktion(H, e)
od
$(C, \bar{C}) \leftarrow$ ⟨*die beiden Mengen (von Knoten von* G*), die den beiden Knoten von* H *entsprechen*⟩

6.13 Satz. *Algorithmus 6.12 kann so implementiert werden, dass seine Laufzeit in* $O(n^2)$ *ist.*

6.14 Beweis. Jeder Aufruf der Funktion Kontraktion benötigt Laufzeit $O(n)$. Bei jedem Schleifendurchlauf wird die Anzahl der Knoten von H um 1 erniedrigt, d. h. es gibt $n - 2$ solche Durchläufe.
Es bleibt zu zeigen, dass der Algorithmus so implementiert werden kann,

1. dass man gleichverteilt zufällig eine Kante des Multigraphen auswählen kann, und
2. dass die Mengen C und $\bar{C}$ hinreichend schnell beschafft werden können.

Das geht zum Beispiel so:

1.
$i \leftarrow$ **random**$(1, 2m)$ ⟨*Beachte: jede Kante wird gleich zweimal gezählt!*⟩
$x \leftarrow 0$; $s \leftarrow 0$
while $s < i$ **do**
  $x \leftarrow x + 1$
  $s \leftarrow s + M[x]$
od
$i \leftarrow i - (s - M[x])$
$y \leftarrow 0$; $s \leftarrow 0$
while $s < i$ **do**
  $y \leftarrow y + 1$
  $s \leftarrow s + A[x, y]$
od
⟨*Wähle Kante zwischen* x *und* y⟩

2. Man führt in der Prozedur Kontraktion zusätzlich eine weitere Datenstruktur mit. Es handelt sich um eine boolesche Matrix Q mit so vielen Zeilen und Spalten wie der ursprüngliche Graph G Knoten hat. Eine 1 als Eintrag in $Q[x, y]$ bedeutet, dass die Knoten, die ursprünglich die Nummern x und y hatten, durch Kontraktionen zusammengefasst wurden. Initialisiert werden muss die Matrix also mit der Einheitsmatrix. Algorithmus 6.11 sollte also unmittelbar vor der letzten Zuweisung $n \leftarrow n - 1$ um die folgenden Zeilen erweitert werden:

 © Th. Worsch 2000-2015

$Q[x, \cdot] \leftarrow Q[x, \cdot] \vee Q[y, \cdot]$
$Q[y, \cdot] \leftarrow Q[n, \cdot]$

Da am Ende sicher nur noch zwei zusammengefasste Knoten existieren, ist die Bestimmung von C und $\bar{C}$ leicht: C ist z. B. durch die 1-Einträge in der ersten Zeile von Q gegeben und $\bar{C}$ durch die 0-Einträge dieser Zeile.

■

6.15 Satz. *Algorithmus 6.12 findet mit Wahrscheinlichkeit $\Omega(n^{-2})$ einen minimalen Schnitt.*

6.16 Beweis. Wir führen den Beweis in drei Schritten:

1. Es sei $(C, \bar{C})$ irgendein Schnitt des ursprünglichen Graphen (der noch keine Mehrfachkanten hat). Wir behaupten, dass Algorithmus 6.12 genau dann diesen Schnitt als Ergebnis liefert, wenn keine der Kanten, die über ihn läuft, während des Algorithmus kontrahiert wird: Es sei e eine am Ende noch vorhandene Kante, „die" ursprünglich zwei Knoten x und y verband. Dann ist nun etwa $x \in C$ und $y \in \bar{C}$. Wäre die Kante e kontrahiert worden, müssten aber x und y am Ende zum gleichen Knoten von H gehören.

2. Es sei $(K, \bar{K})$ ein minimaler Schnitt von G der Größe k. Nach $i-1$ Schleifendurchläufen sei noch keine Kante dieses Schnittes kontrahiert worden. Dann ist also $(K, \bar{K})$ auch ein Schnitt des dann erhaltenen Graphen H_{i-1} und wegen Lemma 6.9 muss es auch ein minimaler Schnitt sein. Also enthält H_{i-1} noch mindestens $(n-i+1)k/2$ Kanten. Folglich ist die Wahrscheinlichkeit, dass als nächstes eine der k Kanten, die zum Schnitt K gehören, kontrahiert wird, höchstens $2/(n-i+1)$ und die Wahrscheinlichkeit, dass keine dieser Kanten kontrahiert wird, mindestens $1-2/(n-i+1)$.

3. Die Wahrscheinlichkeit, dass in keinem der Schritte einer der Kanten des Schnittes K kontrahiert wird, ist daher mindestens

$$\prod_{i=1}^{n-2}\left(1-\frac{2}{n-i+1}\right) = \frac{\prod_{i=1}^{n-2}(n-i-1)}{\prod_{i=1}^{n-2}(n-i+1)} = \frac{(n-2)!}{n!/2} = \frac{(n-2)!2!}{n!} > \frac{2}{n^2} \in \Omega(1/n^2)$$

■

Die Erfolgs-Wahrscheinlichkeit von $2/n^2$ in Satz 6.15 ist natürlich sein klein. Um sie zu vergrößern, kann man den mehrfach ausführen und einen kleinsten der dabei auftretenden Schnitte als Ergebnis wählen.

Wählt man $k = n^2/2$ Wiederholungen, ist die Wahrscheinlichkeit, keinen minimalen Schnitt zu finden, noch

$$\left(1-\frac{2}{n^2}\right)^{n^2/2} < \frac{1}{e} .$$

Weitere Rechnungen zeigen, dass $k \in \Omega(n^2)$ Wiederholungen auch notwendig sind, um konstante Fehlerwahrscheinlichkeit zu erreichen. Die resultierende Gesamtlaufzeit ist damit in $\Theta(n^4)$ im Gegensatz zu $O(n^3)$ für den schnellsten bekannten deterministischen Algorithmus.

Das Problem besteht im Grunde darin, dass die Wahrscheinlichkeit, „aus Versehen" eine Kante des minimalen Schnittes zu kontrahieren zu groß ist, wenn der Graph schon weit kontrahiert wurde. Einen besseren Algorithmus erhält man, wenn randomisiert nur bis zu einem Graphen mit t Knoten kontrahiert, und dann deterministisch weiter arbeitet.

 © Th. Worsch 2000-2015

6.17 Algorithmus. (Beschränkte iterierte Kontraktion)
proc graph ← IterContract(**graph** G, **int** t)
⟨*Eingabe: ein Multigraph* $G(V, E, \nu)$ *und Anzahl* t *von Knoten am Ende*⟩
⟨*Ausgabe: ein kontrahierter Graph* H⟩
$H \leftarrow G$
while (H hat mehr als t Knoten) **do**
e ←⟨*zufällig gleichverteilt gewählte Kante von* H⟩
$H \leftarrow$ Kontraktion(H, e)
od
return H

6.18 Lemma. Dei Wahrscheinlichkeit, dass bei Algorithmus 6.17 im Ergebnisgraphen H noch ein minimaler Schnitt des ursprünglichen Graphen „vorhanden" ist, ist mindestens

$$\binom{t}{2} / \binom{n}{2} .$$

6.19 Beweis. Eine Abschätzung analog wie in Beweis 6.16 ergibt in diesem Fall:

$$\begin{aligned}\prod_{i=1}^{n-t} \left(1 - \frac{2}{n-i+1}\right) &= \frac{\prod_{i=1}^{n-t}(n-i-1)}{\prod_{i=1}^{n-t}(n-i+1)} = \frac{\prod_{i=1}^{n-t-2}(n-i-1)t(t-1)}{n(n-1)\prod_{i=3}^{n-t}(n-i+1)} \\ &= \frac{t(t-1)}{n(n-1)} = \binom{t}{2} / \binom{n}{2} \in \Omega((t/n)^2)\end{aligned}$$

■

6.20 Algorithmus.
proc cut ← IterContractDetMinCut(**graph** G, **int** t)
⟨*Eingabe: ein Multigraph* $G(V, E, \nu)$ *und Anzahl* t *von Knoten für Zwischengraphen*⟩
⟨*Ausgabe: ein Schnitt von* G⟩
$C \leftarrow$ ⟨*ein trivialer Schnitt*⟩
for $i \leftarrow 1$ **to** n^2/t^2 **do**
$H \leftarrow$ IterContract(G, t)
$D \leftarrow$ DetMinCut(H)
$C \leftarrow \min(C, D)$
od
return C

6.21 Lemma. Wählt man in Algorithmus 6.20 $t = n^{2/3}$, dann ist die Laufzeit in $O(n^{8/3})$ und die Wahrscheinlichkeit, einen minimalen Schnitt von G zu erhalten mindestens $1 - 1/e$.

6.22 Beweis. Der Zeitbedarf ist offensichtlich

$$\frac{n^2}{t^2} \cdot (n^2 + t^3) = \frac{n^4}{t^2} + n^2 t .$$

Für den gewählten Wert von t sind beide Summanden größenordungsmäßig in $O(n^{8/3})$.

Die Fehlerwahrscheinlichkeit ist höchstens

$$\left(1 - \frac{t^2}{n^2}\right)^{n^2/t^2} < \frac{1}{e} .$$

■

 © Th. Worsch 2000-2015

Man kann das Problem aber noch schneller lösen. Die Idee besteht darin, für im Laufe der Berechnung erhaltene kontrahierte Zwischengraphen umso häufiger (randomisiert) nach einem minimalen Schnitt zu suchen, je kleiner diese Graphen schon sind.

6.23 ALGORITHMUS. (MINIMALER SCHNITT SCHNELL)

proc cut ← FastCut(**graph** G):
⟨*Eingabe: ein Multigraph* $G(V, E, \nu)$⟩
⟨*Ausgabe: ein Schnitt* C⟩
if ($|V| \leqslant 6$) **then**
  $C \leftarrow$ ⟨*minimaler Schnitt, mittels vollständiger Suche ermittelt*⟩
else
  $t \leftarrow \lceil 1 + n/\sqrt{2} \rceil$
  $H_1 \leftarrow$ IterContract(G, t)
  $H_2 \leftarrow$ IterContract(G, t)
  $C_1 \leftarrow$ FastCut(H_1)
  $C_2 \leftarrow$ FastCut(H_2)
  $C \leftarrow$ ⟨*der kleinere der beiden Schnitte* C_1 *und* C_2⟩
fi

6.24 SATZ. *Algorithmus 6.23 hat Laufzeit* $O(n^2 \log n)$.

6.25 BEWEIS. Die maximale Rekursionstiefe ist $\Theta(\log n)$. Die beiden Aufrufe von **IterContract** benötigen eine Zeit in $O(n^2)$. (In dieser Zeit könnten sogar Kontraktionen auf Graphen mit 2 Knoten durchgeführt werden.) Also ist der Gesamtzeitbedarf $T(n)$ für Eingabegraphen mit n Knoten

$$T(n) = 2 \cdot T\left(\left\lceil 1 + n/\sqrt{2} \right\rceil\right) + O(n^2) \ .$$

Hieraus ergibt sich $T(n) \in O(n^2 \log n)$. ∎

6.26 SATZ. *Algorithmus 6.23 liefert mit einer Wahrscheinlichkeit in* $\Omega(1/\log n)$ *einen minimalen Schnitt.*

6.27 BEWEIS. Es sei G ein Eingabegraph für den Algorithmus, dessen minimale Schnitte die Größe k haben. Ein solcher Schnitt möge eine Reihe von rekursiven Aufrufen von FastCut bis zu einer Stelle „überlebt“ haben; der dann erreichte Graph heiße H. Die durch die Aufrufe von **IterContract** daraus resultierenden Graphen seien H_1 und H_2. Der Aufruf für H wird als Ergebnis einen minimalen Schnitt für G liefern, falls für ein H_i gilt:

1. Der Schnitt überlebt die Kontraktionen zur Konstruktion von H_i.
2. FastCut(H_i) findet einen minimalen Schnitt in H_i.

Die Wahrscheinlichkeit für Punkt 1 ist nach Lemma 6.18 mindestens

$$\frac{\lceil 1 + t/\sqrt{2} \rceil (\lceil 1 + t/\sqrt{2} \rceil - 1)}{t(t-1)} \geqslant \frac{t/\sqrt{2} \cdot t/\sqrt{2}}{t(t-1)} \geqslant \frac{1}{2} \frac{t^2}{t(t-1)} \geqslant \frac{1}{2} \ .$$

Zu Punkt 2.: Wir interessieren uns nun für eine *untere Schranke* $p(r)$ der Wahrscheinlichkeit, dass FastCut r Niveaus über dem Rekursionsabbruch einen minimalen Schnitt findet. Man mache sich klar, dass die wie folgt definierten Werte solche untere Schranken darstellen: $p(0) = 1$ und

$$p(r+1) = 1 - \left(1 - \frac{1}{2} \cdot p(r)\right)^2 = p(r) - \frac{p(r)^2}{4} \ .$$

 © Th. Worsch 2000-2015

Setzt man $q(r) = 4/p(r) - 1$, bzw. $p(r) = 4/(q(r)+1)$, so ergibt sich hieraus

$$\frac{4}{q(r+1)+1} = \frac{4}{q(r)+1} - \frac{4}{(q(r)+1)^2} = \frac{4q(r)}{(q(r)+1)^2}$$

und weiter

$$q(r+1) = q(r) + 1 + \frac{1}{q(r)}$$

Per Induktion kann man leicht zeigen, dass für alle r gilt: $r < q(r) < r + 3 + H_{r-1}$. (Zur Erinnerung: Die r-te harmonische Zahl ist $H_r = \sum_{i=1}^{r} 1/i$.) Denn es gilt:

$$r + 1 < q(r) + 1 < q(r+1) = q(r) + 1 + \frac{1}{q(r)} < r + 3 + H_{r-1} + 1 + \frac{1}{r} = (r+1) + 3 + H_r \; .$$

Daher ist $q(r) \in r + O(\log r)$ und folglich $p(r) \in \Omega(1/r)$. Für einen Ausgangsgraphen mit n Knoten ist aber die Rekursionstiefe $\Theta(\log n)$ und damit die gesuchte Wahrscheinlichkeit $\Omega(1/\log n)$. ■

 © Th. Worsch 2000-2015

6.2 Minimale spannende Bäume

Ausgangspunkt für die in diesem Abschnitt betrachtete Problemstellung sind ungerichtete Graphen, deren Kanten e mit reellen Zahlen $w(e)$ gewichtet sind. Das *Gewicht* $w(T)$ eines Teilgraphen T ist die Summe der Gewichte der Kanten, die zu T gehören.

Mit n wird wieder die Anzahl der Knoten und mit m die Anzahl der Kanten von G bezeichnet.

6.28 PROBLEM. (MINIMUM SPANNING TREES (MST)) Gegeben: ein zusammenhängender ungerichteter Graph $G = (V, E)$, dessen Kanten e mit rellen Zahlen $w(e)$ gewichtet sind.
Gesucht: ein Teilgraph von G, der ein Baum ist, der G aufspannt und unter allen solchen Bäumen minimales Gewicht hat.

Ist der Ausgangsgraph nicht zusammenhängend, dann existiert nur ein minimaler aufspannender Wald (MSF), der aus den MST für die einzelne Zusammenhangskomponenten besteht.

6.29 Man kann ohne Beschränkung der Allgemeinheit davon ausgehen, dass die Gewichte aller Kanten eines Graphen paarweise verschieden sind. Das erreicht man nötigenfalls, indem man alle Kanten $e_1, \dots, e_k$ mit gleichem $w(e_1) = \cdots = w(e_k)$ willkürlich total ordnet und festlegt, dass eine in der Ordnung frühere Kante auch „leichter“ sei.

Die nachfolgend betrachteten Algorithmen haben übrigens die Eigenschaft, dass die tatsächlichen Gewichte irrelevant sind. Es werden immer nur Gewichte zweier Kanten miteinander verglichen, um herauszufinden, welche leichter ist.

6.2.1 Ein deterministischer Algorithmus für MST

Wir bezeichnen im folgenden eine Kante als *lokal minimal*, wenn es einen Knoten gibt, mit dem sie inzidiert und unter allen diesen das kleinste Gewicht hat.

6.30 LEMMA. Jede lokal minimale Kante gehört zu einem MST von G.

6.31 BEWEIS. Es sei e die lokal minimale Kante eines Knotens x; der andere Endpunkt von e sei y. Es sei T' ein aufspannender Baum von G, der *nicht* e enthalte. Wir zeigen: T' hat nicht minimales Gewicht.

Dazu sei e' die Kante von T', die von x wegführt und in T' auf dem kürzesten Weg von x nach y liegt. Man betrachte nun den Graphen T, der aus T' entsteht, indem man e' entfernt und e hinzunimmt. Wir behaupten:

1. $w(T) < w(T')$.
2. T spannt G auf.
3. T ist ein Baum.

Der Reihe nach:

1. Das ist klar, denn $w(e) < w(e')$ und $w(T) = w(T') - w(e') + w(e)$.
2. Es sei z der andere Endpunkt von e' und es seien v_1, v_2 zwei Punkte von G. Wenn der kürzeste Weg in T' zwischen ihnen e' *nicht* beinhaltet, dann sind die beiden in $T' - e'$ immer noch miteinander verbunden. Falls der Pfad e' beinhaltete, dann kann man in T diese Kante ersetzen durch den Pfad von x nach y und von dort zu z. Die beiden Punkte v_1 und v_2 sind also immer noch miteinander verbunden.

© Th. Worsch 2000-2015

3. Angenommen, T enthielte einen Zyklus. Da T' Baum war, muss der Zyklus die Kante e enthalten. Also gibt es in $T' - e'$ einen nicht verkürzbaren Pfad von x nach y. Dieser Pfad ist offensichtlich verschieden vom kürzesten Pfad in T' von x nach y, denn dieser beinhaltete Kante e'. Also gäbe es in T' zwei verschiedene (einer mit e', einer ohne e'), nicht verkürzbare Pfade von x nach y. Also gäbe es in T' auch schon einen Zyklus im Widerspruch dazu, dass T' Baum ist.

■

6.32 KOROLLAR. Die lokal minimalen Kanten eines Graphen bilden ein Wald.

6.33 (BORŮVKA-PHASE) Eine Borůvka-Phase ist ein Algorithmus, der folgendes leistet:

1. Er bestimmt die Menge $L(G)$ aller lokal minimalen Kanten eines Graphen G.
2. Er berechnet den kontrahierten Graphen $G/L(G)$. Falls Mehrfachkanten entstünden, wird nur die mit kleinstem Gewicht behalten und die anderen entfernt.

Der so aus G entstehende Graph werde mit $B(G)$ bezeichnet.

6.34 LEMMA. Eine Borůvka-Phase kann man in Zeit $O(m+n)$ implementieren.

6.35 BEWEIS. Übung. ■

6.36 LEMMA. Durch eine Borůvka-Phase wird die Anzahl der Knoten um mindestens die Hälfte reduziert.

6.37 BEWEIS. Eine Kante kann für höchstens zwei Knoten lokal minimal sein. Also werden in einer Borůvka-Phase mindestens $n/2$ Kanten entfernt. Mit jeder entfernten Kante wird auch ein Knoten entfernt. ■

6.38 LEMMA. Für jeden Graphen G gilt: Die Kanten in $L(G)$ und die Kanten eines MST von $B(G)$ bilden zusammen einen MST von G.

6.39 BEWEIS. Jeder aufspannende Baum T von G, der alle Kanten aus $L(G)$ enthält, induziert einen aufspannenden Baum T' von $B(G)$ mit $w(T') = w(T) - w(L(G))$. Umgekehrt gilt auch, dass jeder aufspannde Baum T' von $B(G)$ einen aufspannenden Baum $\bar{T}'$ von G induziert , der alle Kanten aus $L(G)$ enthält und Gewicht $w(T) = w(T') + w(L(G))$ hat (überlegen Sie sich das).

Sei nun T ein *minimaler* aufspannender Baum von G (er enthält also alle Kanten aus $L(G)$; siehe Lemma 6.30) und $B(T)$ der korrespondierende aufspannende Baum von $B(G)$. Wäre $B(T)$ nicht minimal für $B(G)$, sondern etwa T', so hätte der dadurch induzierte aufspannende Baum $\bar{T}'$ von G nur Gewicht $w(\bar{T}') = w(T') + w(L(G)) < w(B(T)) + w(L(G)) = w(T)$ im Widerspruch zur Minimalität von T. ■

6.40 KOROLLAR. Wenn alle Kantengewichte paarweise verschieden sind, ist der MST eindeutig.

6.41 ALGORITHMUS. (BORŮVKAS MST-ALGORITHMUS) Man führt solange Borůvka-Phasen durch, bis der resultierende Graph höchstens noch zwei Knoten hat und führt Buch, welche Kanten jeweils kontrahiert werden. Wegen Lemma 6.38 ist dann klar, welche Kanten den MST bilden.

6.42 LEMMA. Borůvkas MST-Algorithmus benötigt $O(m \log n)$ Zeit.

6.43 BEWEIS. Nach Lemma 6.36 benötigt man nur $O(\log n)$ viele Phasen, von denen jede nach Lemma 6.34 nur $O(m+n)$ Zeit benötigt. In zusammenhängenden Graphen ist $n \in O(m)$. ■

 © Th. Worsch 2000-2015

6.2.2 F-leichte und F-schwere Kanten

In diesem und im folgenden Abschnitt wird mit F ein Wald in einem Graphen G bezeichnet, d. h. also ein Teilgraph, der aus einem oder mehreren disjunkten Bäumen besteht.

6.44 DEFINITION Es sei F ein Wald in G, v_1 und v_2 zwei Knoten von G und $P(v_1, v_2)$ die Menge der Kanten des (kürzesten) Pfades in F von v_1 nach v_2 (falls er existiert; sonst sei $P = \emptyset$). Wir setzen

$$W_F(v_1, v_2) = \begin{cases} \max\{w(e) \mid e \in P(v_1, v_2)\} & \text{falls } v_1, v_2 \text{ im gleichen Baum von } F \text{ liegen} \\ \infty & \text{sonst} \end{cases}$$

Eine Kante $e = \{v_1, v_2\}$ heißt F-*schwer*, falls $w(e) > W_F(v_1, v_2)$ ist, und sie heißt F-*leicht*, falls $w(e) \leqslant W_F(v_1, v_2)$. ◇

6.45 LEMMA. Es sei F ein beliebiger(!) Wald von G und $e = \{v_1, v_2\}$ eine Kante in G. Wenn e F-schwer ist, dann gehört e nicht zum MST von G.

Äquivalent ist: Wenn eine Kante zum MST gehört, dann ist sie F-leicht.

6.46 BEWEIS. Die Kante $e = \{v_1, v_2\}$ sei F-schwer und es bezeichne P den Pfad von v_1 nach v_2, dessen Kanten alle leichter sind als die Kante e selbst.

Bei der Bestimmung des MST etwa mit Borůvkas Algorithmus wird e *nie* lokal minimale Kante von v_1 oder v_2 sein, sondern immer eine der Kanten von P. Also wird e auch nie zum (eindeutigen; siehe Korollar 6.40) MST hinzugenommen. ■

6.47 LEMMA. Ein aufspannender Baum T eines Graphen G ist minimal, wenn die einzigen T-leichten Kanten in G die Kanten von T sind.

6.48 BEWEIS. Wenn die einzigen T-leichten Kanten in G die Kanten von T sind, dann sind alle Kanten, die nicht zu T gehören T-schwer. Nach Lemma 6.45 gehören sie also sicher nicht zum MST von G. Also besteht der MST nur aus Kanten, die zu T gehören. Von ihnen kann man aber auch keine weglassen, da es dann kein aufspannender Baum von G mehr ist. ■

Ohne einen der (nicht ganz einfachen) Algorithmen (siehe Dixon, Rauch und R. E. Tarjan 1992; King 1997) anzugeben halten wir noch fest:

6.49 SATZ. *Zu einem Graphen G und einem Wald F in G kann man alle F-schweren Kanten von G in Zeit $O(m + n)$ finden.*

6.2.3 Ein randomisierter MSF-Algorithmus

Es sei $\text{RandomSample}(G, p)$ eine Funktion, die einen Graphen G' mit den gleichen Knoten wie in G liefert, bei dem jede Kante von G unabhängig mit Wahrscheinlichkeit p zu G' hinzugenommen wird.

6.50 LEMMA. Zu gegebenen G und p kann ein MSF F' für ein $G' = \text{RandomSample}(G, p)$ konstruiert werden, indem man einmal jede Kante von G betrachtet, sofern die Kanten nach aufsteigendem Gewicht sortiert vorliegen.

 © Th. Worsch 2000-2015

6.51 Beweis. Die Kanten $e_1, \ldots, e_m$ von G seien nach aufsteigendem Gewicht geordnet. Die Konstruktion von $G' = \text{RandomSample}(G, p)$ erfolge, indem nacheinander für jedes i die Kante e_i mit Wahrscheinlichkeit p zu G' hinzu genommen werde.

Die Konstruktion des MSF F' kann in diesem Fall wie folgt erledigt werden:

```
F'_0 ← ∅
k ← 0
for i ← 1 to m do
    if RandomFloat(0,1) ⩽ p then
        ⟨nimm e_i = {u,v} zu G' hinzu⟩
        if u und v gehören zu verschiedenen Zusammenhangskomponenten von F'_k then
            k ← k+1
            F'_k ← F'_{k−1} + e_i
        fi
    fi
od
F' ← F'_k ⟨ist der MSF⟩
```

Offensichtlich wird durch diesen Algorithmus ein Wald konstruiert. Die Bedingung, dass die Enden u und v von e_i zu verschiedenen Zusammenhangskomponenten von F'_k gehören, ist äquivalent zu der Bedingung, dass e_i im betreffenden Schleifendurchlauf gerade F'_k leicht ist.

Nach Lemma 6.47 ist das Endergebnis F' genau dann MSF, falls er jede F'-leichte Kante von G' enthält. Wegen der aufsteigenden Gewichte der Kanten ist ein e von G' genau dann F'-leicht, wenn e zu dem Zeitpunkt, zu dem es auf Zugehörigkeit zu F'_k untersucht wird, F'_k-leicht ist. Und zu diesem Zeitpunkt ist e genau dann F'_k-leicht, wenn es verschiedenen Zusammenhangskomponenten von F'_k verbindet. ■

6.52 Lemma. Es sei F' ein minimaler aufspannender Wald von $G' = \text{RandomSample}(G, p)$. Der Erwartungswert für die Anzahl der F'-leichten Kanten von G (!) ist kleiner oder gleich n/p.

6.53 Beweis. Da der MST eindeutig ist, ist es für die Aussage gleichgültig, mit welchem Algorithmus er berechnet wird.

Betrachten wir den Algorithmus aus Beweis 6.51. Als *Phase* i ($i \geqslant 1$) bezeichnen wir die Reihe Schleifendurchläufe, zu deren Beginn k immer den Wert $i-1$ hat. Salopp gesprochen beginnt Phase i, nachdem F'_{i-1} gebildet wurde und endet mit der Bildung von F'_i.

Während jeder Phase werden sowohl einige F'_{i-1}-schwere Kanten betrachtet, die für den Beweis offensichtlich irrelevant sind, und einige F'_{i-1}-leichte Kanten.

Jede Kante von G, die F'_{i-1}-leicht ist, hat Wahrscheinlichkeit p, dass sie zu G' und folglich zu F'_{i-1} hinzu genommen wird. Und mit der ersten Wahl einer solchen Kante endet die Phase auch. Folglich ist die Anzahl der in Phase i betrachteten F'_{i-1}-leichten Kanten geometrisch verteilt mit Parameter p. Der Erwartungswert für die Anzahl ist $1/p$.

Am Ende ist $|F'| = k \leqslant n-1$. Folglich ist die Zufallsvariable X der bis zum Ende von Phase k betrachteten leichten Kanten die Summe von k identisch geometrisch verteilten unabhängigen Zufallsvariablen. Um die danach noch betrachteten, aber nicht hinzugenommenen Kanten mit zu berücksichtigen, stelle man sich vor, dass die **for**-Schleife noch länger durchlaufen werde, bis insgesamt n mal die Bedingung $\text{RandomFloat}(0,1) < p$ erfüllt war. Der Erwartungswert für die Anzahl Schritte, bis das jeweils einmal der Fall war, ist ebenfalls $1/p$. Es bezeichne Y die Zufallsvariable für die dafür insgesamt notwendige Anzahl von Schleifendurchläufen.

 © Th. Worsch 2000-2015

Dann gilt für alle z: $\mathbf{Pr}[X > z] \leqslant \mathbf{Pr}[Y > z]$ (man sagt, dass X von Y stochastisch dominiert werde). Es gilt dann auch $\mathbf{E}[X] \leqslant \mathbf{E}[Y]$. Dieser Erwartungswert ist aber $\mathbf{E}[Y] = n/p$. ∎

Der Algorithmus aus Beweis 6.51 dient nur dazu, den Beweis von Lemma 6.52 zu erleichtern; dessen Aussage ist aber unabhängig davon, wie man den MSF konstruiert hat. Der Algorithmus ist im Weiteren *nicht* von Bedeutung Es ist also z. B. nicht notwendig, dass die Kanten nach Gewicht sortiert vorliegen.

6.54 Algorithmus.

proc $F \leftarrow \text{MSF}(G)$:
⟨*Eingabe: gewichteter ungerichteter Graph* G *mit* n *Knoten und* m *Kanten*⟩
⟨*Ausgabe: der minimale aufspannende Wald* F *von* G⟩

1. $G_1 \leftarrow B(B(B(G)))$
   $C_1 \leftarrow$ ⟨*die Kanten, die bei der Berechnung von* G_1 *kontrahiert werden*⟩
   if ⟨G_1 *enthält keine Kanten mehr*⟩ **then**
   $F \leftarrow C_1$
   return F
   fi
2. $G_2 \leftarrow \text{RandomSample}(G_1, 1/2)$
3. $F_2 \leftarrow \text{MSF}(G_2)$
4. $C_2 \leftarrow$ ⟨*die* F_2*-schweren Kanten in* G_1⟩
   $G_3 \leftarrow$ ⟨G_1 *ohne die Kanten in* C_2⟩
5. $F_3 \leftarrow \text{MSF}(G_3)$
6. $F \leftarrow C_1 \cup F_3$
   return F

6.55 Satz. *Algorithmus 6.54 berechnet den MSF des Eingabegraphen* G.

6.56 Beweis.

zu 1. Nach Lemma 6.38 ist klar, dass die Kanten aus C_1 zum MSF von G gehören. Enthält G_1 keine weiteren Kanten, dann bilden also die Kanten aus C_1 gerade den MSF.

zu 3. F_2 ist ein Wald in G_2 und folglich auch in G_1 und G.

zu 4. Nach Lemma 6.45 gehören die F_2-schweren Kanten von G_1 nicht zum MSF von G_1. Folglich hat G_3 den gleichen MSF wie G_1,

zu 5. der als F_3 berechnet wird.

zu 6. Wieder nach Lemma 6.38 ist daher $C_1 \cup F_3$ ein MSF und also (wegen Zusammenhang) ein MST von G.

∎

6.57 Satz. *Der Erwartungswert für die Laufzeit von Algorithmus 6.54 ist* $O(m+n)$.

6.58 Beweis. Es bezeichne $T(n, m)$ den Erwartungswert der Laufzeit von Algorithmus 6.54 für Eingabegraphen mit n Knoten und m Kanten. Für die einzelnen Schritte gilt:

1. Laufzeit ist $O(m+n)$. Und G_1 hat höchstens $n/8$ Knoten und m Kanten.

© Th. Worsch 2000-2015

2. Laufzeit ist $O(m+n)$. Und G_2 hat höchstens $n/8$ Knoten und der Erwartungswert für die Anzahl Kanten ist $m/2$.
3. Erwartungswert für die Laufzeit ist $T(n/8, m/2)$.
4. Laufzeit ist $O(m+n)$. Und G_3 hat höchstens $n/8$ Knoten und Erwartungswert für die Anzahl Kanten ist $(n/8)/(1/2) = n/4$ (nach Lemma 6.52).
5. Erwartungswert für die Laufzeit ist $T(n/8, n/4)$.
6. Laufzeit ist $O(n)$.

Insgesamt ergibt sich

$$T(n, m) \leqslant T(n/8, m/2) + T(n/8, n/4) + c(n+m) .$$

Als Lösung dieser Rekurrenzungleichung ergibt sich $T(n, m) \in O(n+m)$. ∎

Literatur

Dixon, B., M. Rauch und R. E. Tarjan (1992). „Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time“. In: *SIAM Journal on Computing* 21.6, S. 1184–1192 (siehe S. 57).

King, Valerie (1997). „A Simpler Minimum Spanning Tree Verification Algorithm“. In: *Algorithmica* 18.2, S. 263–270 (siehe S. 57).

 © Th. Worsch 2000-2015

7 Random Walks

Wir beginnen mit einem motivierenden Beispiel.

7.1 Ein randomisierter Algorithmus für 2-SAT

Bekanntlich ist 3-SAT ein **NP**-vollständiges Problem. Hingegen ist 2-SAT auch deterministisch in Polynomialzeit lösbar. Hierbei kommen also in jeder Klausel nur *zwei* Literale vor. Wir wollen im folgenden einen randomisierten Algorithmus für 2-SAT betrachten.

Im folgenden sei F eine erfüllbare Formel mit n Variablen (*nicht* Vorkommen von Literalen). Wie man den nachfolgenden Algorithmus modifizieren sollte, damit auch nicht erfüllbare Formeln sinnvoll behandelt werden, wird in Punkt 7.4 kurz diskutiert.

7.1 Algorithmus.

$B \leftarrow$ ⟨*zufällige Belegung aller* $x_1, \dots, x_n$ *mit Werten*⟩
while $F(B) =$ *false* **do**
  $k \leftarrow$ ⟨*von* B *nicht erfüllte Klausel in* F⟩
  $x_i \leftarrow$ ⟨*zufällig gewählte Variable in* k⟩
  $B(x_i) \leftarrow$ **not** $B(x_i)$
od

7.2 Für die Analyse des Algorithmus sei A eine fixierte Variablenbelegung, die F erfüllt. (Im allgemeinen kann es mehrere erfüllende Belegungen geben.) Mit j werde die Anzahl Variablen x bezeichnet, für die $B(x) = A(x)$ ist.

Betrachtet man den linearen Pfad P_{n+1}, dessen Knoten die möglichen Werte $0, 1, \dots, n$ von j sind, so entspricht die jeweils aktuelle Belegung B des Algorithmus einem Knoten. Bei jedem Schleifendurchlauf wird j entweder um 1 erhöht oder um 1 erniedrigt, d. h. jede Veränderung von B entspricht dem Schritt zu einem der beiden Nachbarknoten.

Im schlimmsten Fall beginnt der Algorithmus am Knoten $j = 0$ und endet spätestens dann, wenn zum ersten mal $j = n$ erreicht wird. (Das Ende kann auch schon vorher erreicht sein, wenn es außer A noch andere erfüllende Belegungen gibt.)

7.3 Eine Erhöhung der Knotennummer findet mindestens mit Wahrscheinlichkeit $1/2$ statt. Die Frage nach der Laufzeit des Algorithmus ist gleichbedeutend mit der Frage nach der Anzahl Schritte vom Startknoten bis zum Finden einer erfüllenden Belegung. Diese kann nach oben abgeschätzt werden durch die Anzahl Schritte von Knoten 0 zu Knoten n.

Das ist eine typische Fragestellung bei *Random Walks*: Was ist der Erwartungswert für die Anzahl Schritte, um von einem bestimmten Startknoten zu einem bestimmten Zielknoten zu gelangen?

Bevor wir genauer auf Random Walks eingehen, teilen wir schon ein mal mit die Antwort auf die oben angeschnittenen Fragen mit:

7.4 Der Erwartungswert für die Anzahl Schritte von Knoten 0 nach Knoten n entlang eines einzelnen Pfades ist $\leqslant n^2$ (vgl. die Argumentation in Beispiel 7.13).

Wegen der Markov-Ungleichung ist daher die Wahrscheinlichkeit, dass ein Random Walk der Länge $2n^2$ *nicht* zum Ziel führt höchstens $1/2$. Das motiviert die folgende Variante des Algorithmus:

$B \leftarrow$ ⟨*zufällige Belegung aller* $x_1, \dots, x_n$ *mit Werten*⟩
$m \leftarrow 0$ ⟨*Zähler für die Anzahl Versuche*⟩
while $F(B) =$ *false* **and** $m < 2n^2$ **do**
    $k \leftarrow$ ⟨*von* B *nicht erfüllte Klausel in* F⟩
    $x_i \leftarrow$ ⟨*zufällig gewählte Variable in k*⟩
    $B(x_i) \leftarrow$ **not** $B(x_i)$
    $m \leftarrow m + 1$
od
if $F(B) =$ *true* **then**
    return ⟨F *erfüllbar durch* B⟩
else
    return ⟨F *nicht erfüllbar*⟩
fi

7.2 Random Walks

7.5 In diesem Kapitel bezeichne $G = (V, E)$ einen endlichen, zusammenhängenden, ungerichteten Graphen mit $|V| = n \geqslant 2$ Knoten und $|E| = m$ Kanten. Für einen Knoten u bezeichnen wir mit $\Gamma(u) = \{v \mid (u, v) \in E\}$ die Menge der „Nachbarn“, so dass $d(u) = |\Gamma(u)|$ der Grad von u ist.

Wir stellen uns vor, dass ein Objekt, Agent, Random Walker, Irrläufer, oder wie auch immer man es oder ihn nennen möchte, auf die folgende Art „auf dem Graphen herumläuft“:

- Zu jedem Zeitpunkt befindet er sich an einem Knoten des Graphen, und
- jeder Schritt besteht darin, dass sich der Random Walker von seinem aktuellen Knoten zufällig zu einem anderen bewegt, zu dem eine Kante führt, wobei jede ausgehende Kante mit gleicher Wahrscheinlichkeit gewählt wird.

7.6 Definition

- Mit m_{uv} wird der Erwartungswert bezeichnet für die Anzahl Schritte, die benötigt wird, um von Knoten u erstmals zu Knoten v zu gelangen.
- Die *Wechselzeit* $C_{uv} = m_{uv} + m_{vu}$ ist die erwartete Anzahl Schritte für einen Random Walk, der von u nach u führt und unterwegs mindestens einmal v besucht. ◇

Das folgende Beispiel zeigt, dass man z. B. bei m_{uv} mit intuitiv erscheinenden Annahmen vorsichtig sein muss.

7.7 Beispiel. Es bezeichne $L_n = (V, E)$ den Graphen mit Knotenmenge $V = \{1, \dots, n\}$, bei dem die ersten $n/2$ Knoten eine Clique bilden, an die ein Pfad mit den anderen $n/2$ Knoten „angeklebt“ ist, also $E = \{((u, v) \mid 1 \leqslant u, v \leqslant n/2\} \cup \{(u, u+1) \mid n/2 \leqslant u < n\}$.

Es bezeichne nun u den Knoten $n/2$ und v den Knoten n (siehe Abbildung 7.1 für ein Beispiel).

 © Th. Worsch 2000-2015

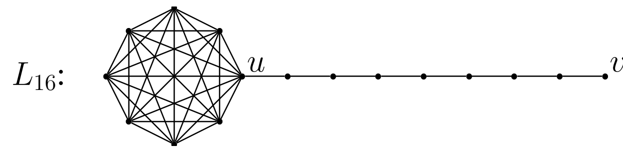


Abbildung 7.1: Der Graph L_{16}.

Wie wir weiter unten sehen werden, ist $m_{uv} \in \Theta(n^3)$ während $m_{vu} \in \Theta(n^2)$ ist. Die beiden Werte können also „weit“ auseinander liegen. (Man versuche, sich schon einmal klar zu machen, woher der Unterschied kommt!)

7.3 Widerstandsnetzwerke

7.8 Jedem ungerichteten zusammenhängenden Graphen G ohne Schlingen entspricht ein Netzwerk $N(G)$ elektrischer Widerstände, indem man jede Kante von G durch einen Widerstand von 1 Ω ersetzt.

Für zwei verschiedene Knoten u und v in $N(G)$ kann man den *effektiven Widerstand* R_{uv} zwischen ihnen bestimmen. Das ist (natürlich) der Quotient U_{uv}/I_{uv} aus einer zwischen u und v angelegten Spannung und dem dann fließenden Strom.

Für die einfache Reihen- bzw. Parallelschaltung von Widerständen R_k hat man vermutlich schon früher gelernt, dass der sich ergebende Gesamtwiderstand $R = \sum_k R_k$ bzw. $R = 1/(\sum_k 1/R_k)$ ist. In allgemeineren Fällen ist die Bestimmung etwas schwieriger. Was ist z. B. der effektive Widerstand zwischen raumdiagonal gegenüberliegenden Ecken eines Würfels?

Der folgende Satz zeigt, warum wir uns hier für Widerstandsnetzwerke interessieren:

7.9 SATZ. *Für beliebige Knoten u und v in G gilt:* $C_{uv} = 2mR_{uv}$.

Als vorbereitenden Schritt zeigen wir einen Zusammenhang von m_{uv} mit einem elektrischen Sachverhalt:

7.10 LEMMA. Für zwei Knoten u und v von G bezeichne φ_{uv} die Spannung bei u relativ zu v, die man misst, wenn man in $N(G)$ jedem Knoten x ein Strom von $d(x)$ Ampere zugeführt wird und der gesamte Strom (von insgesamt $2m$ Ampere) an Knoten v abgeführt wird. Dann ist $m_{uv} = \varphi_{uv}$.

In Abbildung 7.2 sind für einen Beispielgraphen alle Ströme und φ_{uv} dargestellt. Die Stromstärke ist durch die Pfeillänge wiedergegeben; zufließende Ströme sind blau und der abfließende rot dargestellt.

7.11 BEWEIS. Da alle Widerstände 1 Ω sind, ist der von einem Knoten $u \in V \setminus \{v\}$ zu einem Nachbarknoten $w \in \Gamma(u)$ fließende Strom gerade $\varphi_{uv} - \varphi_{wv}$. Nach der Kirchhoffschen Regel muss für jeden Knoten u der dort zufließende Strom gleich dem von dort abfließenden Strom sein. Also ist

$$d(u) = \sum_{w \in \Gamma(u)} (\varphi_{uv} - \varphi_{wv}) \qquad \text{bzw.} \qquad d(u) + \sum_{w \in \Gamma(u)} \varphi_{wv} = d(u)\varphi_{uv} \,.$$

 © Th. Worsch 2000-2015

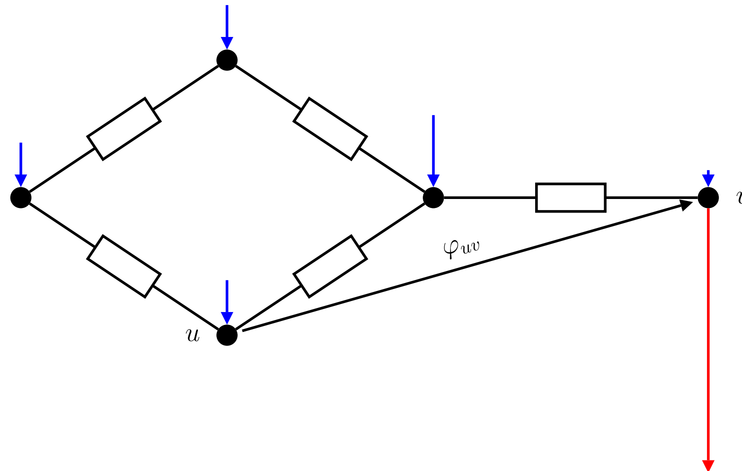


Abbildung 7.2: Ströme für die Messung von φ_{uv}.

Andererseits gilt wegen der Linearität des Erwartungswertes für $u \in V \setminus \{v\}$:

$$m_{uv} = \sum_{w \in \Gamma(u)} (1 + m_{wv})/d(u) \qquad \text{bzw.} \qquad d(u) + \sum_{w \in \Gamma(u)} m_{wv} = d(u) m_{uv} \ .$$

Wie man sieht, liegt hier zweimal das gleiche lineare Gleichungssystem vor. Es hat auch eine eindeutige Lösung. (Wer das genauer dargelegt haben will, werfe einen Blick auf die Vorlesungsfolien oder lese die schöne Arbeit von Doyle und Snell (2000).) Also gilt $m_{uv} = \varphi_{uv}$. ∎

7.12 Beweis. (von Satz 7.9) Aus dem vorangegangenen Lemma wissen wir bereits, dass $m_{uv} = \varphi_{uv}$ ist (Abb. 7.3 (a)). Analog ist $m_{vu} = \varphi_{vu}$ die Spannung bei v relativ zu u, wenn dem Widerstandsnetzwerk an jedem Knoten x ein Strom von $d(x)$ Ampere zugeführt wird und der gesamte Strom (von insgesamt $2m$ Ampere) an Knoten u abgeführt wird (Abb. 7.3 (b)). Umdrehen aller Vorzeichen ergibt, dass $m_{vu} = \varphi_{vu}$ auch die Spannung bei u relativ zu v ist, wenn aus dem Widerstandsnetzwerk an jedem Knoten x ein Strom von $d(x)$ Ampere abgeführt wird und der gesamte Strom (von insgesamt $2m$ Ampere) an Knoten u zugeführt wird (Abb. 7.3 (c)).

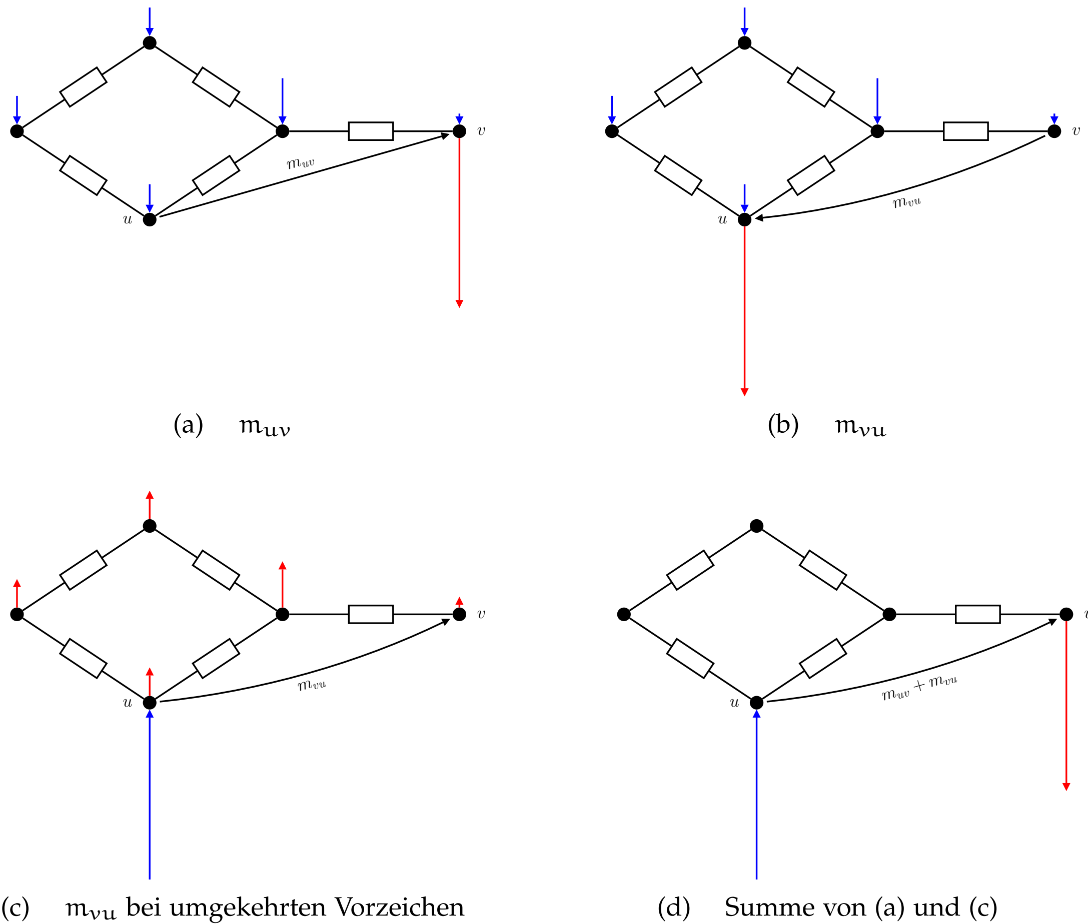
Widerstandsnetzwerke sind linear. Also ist $C_{uv} = m_{uv} + m_{vu} = \varphi_{uv} + \varphi_{vu}$ die Spannung bei u relativ zu v, wenn an jedem Knoten x jeweils $d(x)$ Ampere zu- und abgeführt werden (also kein Strom zu- oder abgeführt wird) und außerdem an Knoten u $2m$ Ampere zu- und an Knoten v $2m$ Ampere abgeführt werden (Abb. 7.3 (d)).

Nach dem Ohmschen Gesetz ist diese Spannung aber gerade $2mR_{uv}$. ∎

7.13 Beispiel. Mit Hilfe der Charakterisierung von m_{xy} mittels φ_{xy} kann man z. B. für den Graphen L_n aus Beispiel 7.7 die Werte m_{uv} und m_{vu} bestimmen.

m_{uv}: An jedem der ersten $n/2$ Knoten werden $n/2$ Ampere injiziert. Dieser Gesamtstrom von $\Theta(n^2)$ Ampere fließt über u und weitere $n/2$ Knoten nach v und verursacht an allen

 © Th. Worsch 2000-2015



(a) m_{uv} (b) m_{vu}

(c) m_{vu} bei umgekehrten Vorzeichen (d) Summe von (a) und (c)

Abbildung 7.3: Die Beweisschritte am Beispiel eines Graphen mit 5 Knoten

Widerständen unterwegs jeweils einen Spannungsabfall von $\Theta(n^2)$ Volt. Diese summieren sich also zu $\Theta(n^3)$.

m_{vu}: An jedem der Knoten $1 + n/2, 2 + n/2, \dots, n-1$ werden 2 Ampere injiziert, die zu Knoten u fließen. Also fließen durch den Widerstand zwischen Knoten $n-i$ und $n-i-1$ $(0 \leqslant i \leqslant n/2+1)$ gerade $2(i+1)$ Ampere. Die entsprechenden Spannungsabfälle summieren sich also zu $\Theta(n^2)$.

7.14 Korollar. Für jeden Graphen mit n Knoten und beliebige Knoten u und v gilt: $C_{uv} < n^3$.

7.15 Beweis. Ein ungerichteter Graph mit n Knoten hat höchstens $m \leqslant n(n-1)/2$ Kanten. Der maximale effektive Widerstand R_{uv} ist nach oben beschränkt durch die Länge der kürzesten Wege von u nach v. Reihenschaltung ist der schlimmste Fall (man überlege sich das). Dann ist $R_{uv} \leqslant n-1$ und folglich insgesamt $C_{uv} = 2mR_{uv} < n^3$. ■

 © Th. Worsch 2000-2015

7.4 Randomisierte Algorithmen für Zusammenhangstests

7.16 Wir betrachten nun das Problem, für einen vorgegebenen ungerichteten Graphen und zwei seiner Knoten festzustellen, ob sie zu der gleichen Zusammenhangskomponente gehören.

Dieses Problem wird üblicherweise mit USTCON bezeichnet (engl. *undirected* s–t *connectivity*). Die Aufgabe besteht mit anderen Worten darin, für zwei beliebige Knoten s und t eines ungerichteten Graphen festzustellen, ob sie durch einen Pfad miteinander verbunden sind.

Dieses Problem taucht im Zusammenhang mit mehreren Problemen aus der Graphentheorie auf. Ohne genauer darauf einzugehen, sei auch noch mitgeteilt, dass es auch in der Komplexitätstheorie von Bedeutung ist. (USTCON ist vollständig für **SL**. Für diese Klasse hat Reingold (2005) gezeigt: **SL** = L.)

7.17 Algorithmus. Wir beschreiben einen randomisierten Algorithmus A, der USTCON mit logarithmischem Platz (und in polynomialer Zeit) mit einseitigem Fehler „löst“: Es wird ein Random Walk auf dem eingegebenen Graphen simuliert, der bei s startet, und zwar maximal $2n^3$ Schritte. Wenn man dabei auf Knoten t trifft, wird sofort YES ausgegeben. Geschieht das nicht, wird am Ende NO ausgegeben.

Dass Algorithmus 7.17 logarithmischen Platz und polynomiale Zeit benötigt, sollte klar sein.

7.18 Lemma. Die Wahrscheinlichkeit, dass A aus Algorithmus 7.17 fälschlicherweise NO ausgibt, ist höchstens 1/2.

7.19 Beweis. Es ist nur der Fall zu betrachten, dass s und t in der gleichen Zusammenhangskomponente liegen. Wegen Korollar 7.14 ist offensichtlich der Erwartungswert $m_{st} \leqslant n^3$. Nach der Markov-Ungleichung aus Satz 4.10 ist daher die Wahrscheinlichkeit, dass ein Random Walk mehr als doppelt solange benötigt, um von s nach t zu gelangen, höchstens 1/2. ■

7.20 Zum Abschluss dieses Kapitels wollen wir noch die Variante des obigen Problems für gerichtete Graphen betrachten. Sie wird üblicherweise mit STCON bezeichnet (engl. s–t *connectivity*). Die Aufgabe besteht darin, für zwei beliebige Knoten s und t eines gerichteten Graphen mit anderen Worten darin festzustellen, ob sie durch einen Pfad miteinander verbunden sind.

Die offensichtliche Anpassung von Algorithmus 7.17 an den gerichteten Fall ist ungeeignet: Der zu untersuchende Graph kann „Sackgassen“ enthalten, aus denen man mit dem Random Walk nicht mehr herauskommt.

Dies ist aber sozusagen das einzige Problem, dem man entkommt, indem man auf geeignete, noch genau zu spezifizierende Weise gelegentlich an den Ausgangspunkt s zurückspringt, wenn man noch nicht t erreicht hat.

7.21 Algorithmus. Es werden abwechselnd immer wieder die beiden folgenden Phasen ausgeführt, bis der Algorithmus in einer von ihnen anhält:

1. Ausgehend von s wird in G ein Random Walk der Länge maximal $n - 1$ durchgeführt. Wird dabei t erreicht, hält der Algorithmus sofort mit der Ausgabe YES an.

2. Es werden $\log n^n = n \log n$ Zufallsbits „gewürfelt“. Wenn sie alle 0 sind, hält der Algorithmus sofort mit der Ausgabe NO an.

 © Th. Worsch 2000-2015

7.22 Lemma.

a) Algorithmus 7.21 kann so implementiert werden, dass der Platzbedarf stets kleiner gleich $O(\log n)$ ist.
b) Wenn kein Pfad von s nach t existiert, gibt der Algorithmus nie YES aus, also stets NO, sofern er hält.
c) Wenn ein Pfad von s nach t existiert, gibt der Algorithmus mit einer Wahrscheinlichkeit größer gleich $1/2$ die Antwort YES aus.

7.23 Beweis.

a) In Phase 1 des Algorithmus muss man im wesentlichen speichern, wie lang der bisherige Random Walk war und bei welchem Knoten man sich gerade befindet. Dafür reichen $O(\log n)$ Bits.

   Phase 2 kann implementiert werden, indem man zählt, wieviele Bits bislang gewürfelt wurden, und ob alle gleich 0 waren. Für den Zähler genügen $\log(n \log n) \in O(\log n)$ Bits.
b) Wenn kein Pfad existiert, wird kein Random Walk von s nach t führen, also wird auch nie YES ausgegeben.
c) Angenommen, es gibt einen Pfad von s nach t. Da es insgesamt im Graphen höchstens n^n Pfade der Länge $n-1$ geben kann, ist die Wahrscheinlichkeit, dass in Phase 1 ein Pfad von s nach t gefunden wird, mindestens n^{-n}. Die Wahrscheinlichkeit, dass in Phase 2 die falsche Antwort gegeben wird, ist $(1-n^{-n})n^{-n} \leqslant n^{-n}$.

   Es sei X die Zufallsvariable, die angibt, in welchem Durchlauf die richtige Antwort gegeben wird, und p die Wahrscheinlichkeit, dass überhaupt die richtige Antwort gegeben wird. Dann ist

$$p = \mathbf{Pr}[X \geqslant 1] = \mathbf{Pr}[X = 1] + \mathbf{Pr}[X \geqslant 2] \geqslant n^{-n} + (1-n^{-n})^2 p \geqslant n^{-n} + (1-2n^{-n})p \, .$$

   Auflösen nach p ergibt sofort $p \geqslant 1/2$.

■

Zusammenfassung

1. Es gibt Zusammenhänge zwischen Random Walks und elektrischen Widerstandsnetzwerken.

2. Random Walks kann man benutzen, um in Graphen zwei Knoten auf Verbundenheit zu testen.

Literatur

Doyle, Peter G. und J. Laurie Snell (2000). *Random walks and electric networks*. arXiv preprint available at the URL: http://arxiv.org/abs/math.PR/0001057 (siehe S. 64).

 © Th. Worsch 2000-2015

Reingold, Omer (2005). „Undirected ST-connectivity in log-space“. In: *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*. Hrsg. von Harold N. Gabow und Ronald Fagin. ACM, S. 376–385. ISBN: 1-58113-960-8 (siehe S. 66).

 © Th. Worsch 2000-2015

8 Markov-Ketten

8.1 Grundlegendes zu Markov-Ketten

Eine Markov-Kette ist ein stochastischer Prozess, der in diskreten Zeitschritten abläuft. Dabei wird jeweils von einem Zustand in einen nächsten übergegangen.

In diesem Abschnitt sind nur einige grundlegende Definitionen und Tatsachen (meist ohne Beweise) zusammengestellt. Ausführlicheres findet man zum Beispiel in den Büchern von Behrends (2000) und in dem auch Online unter `http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book.html` verfügbaren Werk von Grinstead und Snell (1997), sowie etwa bei Feller (1968) und Kemeny und Snell (1976).

8.1 Definition Eine endliche *Markov-Kette* ist durch ein Paar $M = (S, \mathbf{P})$ festgelegt. Dabei bezeichnet S eine endliche Menge von *Zuständen* und $\mathbf{P}$ eine zeilenstochastische $S \times S$-Matrix, die die *Übergangswahrscheinlichkeiten* enthält. (Für alle $i, j \in S$ ist also $0 \leqslant P_{ij} \leqslant 1$ und $\sum_j P_{ij} = 1$.) ◇

Für alle $i, j \in S$ ist P_{ij} als Wahrscheinlichkeit zu interpretieren, dass M vom Zustand i in den Zustand j übergeht. Diese Wahrscheinlichkeit hängt also nur von i ab, und nicht etwa von vorher durchlaufenen Zuständen oder davon, der wievielte Schritt ausgeführt wird.

Im allgemeinen erlaubt man bei Markov-Ketten auch abzählbar unendlich große Zustandsmengen. *Dieser Fall ist im Folgenden stets ausgeschlossen.*

8.2 Im folgenden bezeichnet X_t stets die Zufallsvariable, die den Zustand einer Markov-Kette zum Zeitpunkt t angibt. Es ist also $\mathbf{Pr}[X_{t+1} = j \mid X_t = i] = P_{ij}$. Auch der Anfangszustand X_0 ist im allgemeinen nicht deterministisch, sondern gemäß einer Verteilung festgelegt.

8.3 Ist $\mathbf{q}$ ein Vektor, der zu einem Zeitpunkt t angibt, mit welcher Wahrscheinlichkeit $\mathbf{q}_i$ eine Markov-Kette in Zustand i ist, dann ist $\mathbf{qP}$ der entsprechende Vektor für Zeitpunkt $t+1$, denn es gilt:

$$\mathbf{Pr}[X_{t+1} = j] = \sum_i \mathbf{Pr}[X_t = i]\,\mathbf{Pr}[X_{t+1} = j \mid X_t = i] = \sum_i \mathbf{q}_i P_{ij} = (\mathbf{qP})_j \ .$$

Analog ergibt sich $\mathbf{qP}^t$ für die Verteilung nach t Schritten.

Wir schreiben $P_{ij}^{(t)}$ für die Wahrscheinlichkeit, dass die Markov-Kette in t Schritten von Zustand i in Zustand j übergeht. Es ist also $P_{ij}^{(t)} = (\mathbf{P}^t)_{ij}$.

8.4 Definition

- Eine nichtleere Teilmenge $C \subseteq S$ von Zuständen einer endlichen Markov-Kette heißt *abgeschlossen*, falls für alle $i \in C$ und alle $j \in S \setminus C$ gilt: $P_{ij} = 0$.
- Eine abgeschlossene Teilmenge C heißt *irreduzibel*, falls keine echte (nichtleere) Teilmenge von C auch abgeschlossen ist.
- Eine Markov-Kette heiße *irreduzibel*, falls S als abgeschossene Teilmenge irreduzibel ist. ◇

Jede Markov-Kette besitzt mindestens eine abgeschlossene Teilmenge, nämlich $C = S$.

8.5 Definition Es seien $C_1, \ldots, C_r$ alle irreduziblen Teilmengen einer Markov-Kette S und $T = S \setminus (C_1 \cup \cdots \cup C_r)$. Die Zustände in T (sofern welche existieren) heißen *transient* und die Zustände in den C_k *rekurrent* (oder auch *persistent*). ◇

8.6 Definition

- Die Wahrscheinlichkeit, ausgehend von Zustand i nach t Schritten *erstmals* in Zustand j überzugehen, werde mit $f_{ij}^{(t)} = \mathbf{Pr}[X_t = j \wedge \forall\, 1 \leqslant s \leqslant t-1 : X_s \neq j \mid X_0 = i]$ bezeichnet.
- Die Wahrscheinlichkeit f_{ij}^*, ausgehend von Zustand i irgendwann Zustand j zu erreichen, ist $f_{ij}^* = \sum_{t>0} f_{ij}^{(t)}$.
- Der Erwartungswert m_{ij} für die benötigte Anzahl Schritte, um ausgehend von Zustand i irgendwann zum ersten Mal Zustand j zu erreichen, kann definiert werden als

$$m_{ij} = \begin{cases} \sum_{t \geqslant 1} t \cdot f_{ij}^{(t)} & \text{falls } f_{ij}^* = 1 \\ \infty & \text{sonst} \end{cases}$$

Man beachte, dass auch im Fall $f_{ij}^* = 1$ immer noch $m_{ij} = \infty$ sein kann. ◇

8.7 Jedem Random Walk auf einem Graphen G, wie wir ihn im vorangegangenen Kapitel betrachtet haben, entspricht offensichtlich eine endliche Markov-Kette M_G, indem man $P_{ij} = 0$ setzt, falls keine Kante von i nach j existiert, und andernfalls $P_{ij} = 1/d(i)$, wobei $d(i)$ der Ausgangsgrad des Knotens i ist.

Umgekehrt entspricht jeder Markov-Kette M ein Graph G_M, dessen Knoten die Zustände $i \in S$ sind und in dem eine Kante von i nach j genau dann vorhanden ist, wenn $P_{ij} > 0$ ist. In diesem Fall kann man sich auch noch die Kante mit P_{ij} gewichtet vorstellen.

Man kann nun zeigen:

8.8 Lemma. Es sei eine endliche Markov-Kette gegeben. Dann gilt: Ein Zustand i ist genau dann transient, wenn eine der folgenden (äquivalenten) Bedingungen erfüllt ist:

- $f_{ii}^* < 1$.
- $\sum_{t \geqslant 0} P_{ii}^{(t)} < \infty$.
- Ein Random Walk, der in i startet, kehrt mit Wahrscheinlichkeit 0 unendlich oft nach i zurück.

Analog ergibt sich:

8.9 Lemma. Es sei eine endliche Markov-Kette gegeben. Dann gilt: Ein Zustand i ist genau dann rekurrent, wenn eine der folgenden (äquivalenten) Bedingungen erfüllt ist:

- $f_{ii}^* = 1$.
- $\sum_{t \geqslant 0} P_{ii}^{(t)} = \infty$.
- Ein Random Walk, der in i startet, kehrt mit Wahrscheinlichkeit 1 unendlich oft nach i zurück.

8.10 Definition Ein rekurrenter Zustand i einer Markov-Kette heißt

- *positiv persistent*, falls $m_{ii} < \infty$ ist, und
- *null-persistent*, falls $m_{ii} = \infty$ ist. ◇

 © Th. Worsch 2000-2015

8.2 Irreduzible und ergodische Markov-Ketten

8.11 Für die Anwendungen in dieser Vorlesung sind vor allem irreduzible Markov-Ketten interessant. In diesem Fall ist die gesamte Kette die einzige irreduzible Teilmenge von Zuständen und es gibt also gar keine transienten Zustände.

8.12 Definition Die *Periode* d_i eines Zustandes i ist der größte gemeinsame Teiler aller Zahlen der Menge $N_i = \{t \mid P_{ii}^{(t)} > 0 \land t \in \mathbb{N}_+\}$. Ein Zustand mit Periode 1 heißt auch *aperiodisch*.

Ein Zustand, der aperiodisch und positiv persistent ist, heißt auch *ergodisch*. ◇

8.13 Definition Eine Markov-Kette ist *aperiodisch*, wenn alle ihre Zustände aperiodisch sind. Eine irreduzible und aperiodische Markov-Kette heißt auch *ergodisch*. ◇

Man beachte, dass für aperiodische Zustände *nicht* gilt, dass für alle t automatisch $P_{ii}^{(t)} > 0$ ist. Man kann aber zeigen:

8.14 Lemma. Es sei $M \subseteq \mathbb{N}$ eine Menge natürlicher Zahlen mit der Eigenschaft, dass $M + M = \{k + \ell \mid k, \ell \in M\} \subseteq M$ und $\gcd M = 1$. Dann gibt es ein $k_0 \in \mathbb{N}$ mit $\{k_0, k_0 + 1, k_0 + 2, \dots\} \subseteq M$, d. h. M enthält ab irgendeinem k_0 *alle* natürlichen Zahlen.

Bei den uns in dieser Vorlesung interessierenden Anwendungen von Markov-Ketten kann man sich mitunter auf aperiodische beschränken. Wir skizzieren im folgenden zunächst, warum. Insofern ist es für den weiteren Verlauf der Vorlesung „in Ordnung“, wenn man vor allem an aperiodische Markov-Ketten denkt.

8.15 Ist eine Markov-Kette M mit Matrix $\mathbf{P}$ nicht aperiodisch, dann kann man daraus wie folgt eine neue, aperiodische Markov-Kette M' konstruieren: In M werden alle Wahrscheinlichkeiten mit $1/2$ multipliziert und für jeden Zustand i die Wahrscheinlichkeit P_{ii} um $1/2$ erhöht. Mit anderen Worten ist $\mathbf{P}' = \frac{1}{2}(\mathbf{I} + \mathbf{P})$. ($\mathbf{I}$ bezeichne die Einheitsmatrix.)

Bei dieser Vorgehensweise bleiben einige „Dinge“ erhalten. Zum Beispiel gilt: Ist $\mathbf{wP} = \mathbf{w}$, dann ist auch $\mathbf{wP}' = \mathbf{w}$ und umgekehrt. Allgemeiner haben sogar die beiden Matrizen die gleichen Eigenvektoren. Und aus einem Eigenwert λ von $\mathbf{P}$ wird ein Eigenwert $1/2 + \lambda/2$ von $\mathbf{P}'$. Wir werden im Folgenden sehen, warum gerade das interessant ist.

8.16 Satz. *Es sei* $\mathbf{P}$ *die Matrix einer ergodischen Markov-Kette. Dann gilt:*

- $\mathbf{W} = \lim_{t\to\infty} \mathbf{P}^t$ *existiert.*
- $\mathbf{W}$ *besteht aus identischen Zeilen* $\mathbf{w}$.
- *Alle Einträge von* $\mathbf{w} = (w_1, \dots, w_n)$ *sind echt größer* 0 *und* $\sum_{i=1}^n w_i = 1$.

8.17 Beweis. Da die Markov-Kette ergodisch ist, folgt aus Lemma 8.14, dass es eine Potenz $\mathbf{P}^k$ gibt, deren Einträge *alle echt* größer Null sind. Um die Notation zu vereinfachen nehmen wir im folgenden einfach an, dass schon $\mathbf{P}$ diese Eigenschaft habe. Ansonsten müsste man im Folgenden statt dessen immer mit $\mathbf{P}^k$ arbeiten.

Sei nun zunächst $\mathbf{y}$ ein beliebiger Vektor.

1. Wir zeigen zunächst: Ist $d > 0$ der kleinste in $\mathbf{P}$ vorkommende Eintrag und sind m_0 und M_0 der kleinste resp. der größte Wert eines Vektors $\mathbf{y}$ und m_1 und M_1 der kleinste resp. der größte Wert von $\mathbf{Py}$, dann gilt: $M_1 - m_1 \leqslant (1 - 2d)(M_0 - m_0)$.

   Die Einträge jeder Zeile von P addieren sich zu 1. Für jedes i ist $(\mathbf{Py})_i = \sum_j P_{ij} y_j$. Offensichtlich ist

 © Th. Worsch 2000-2015

- $m_1 = \min_i \sum_j P_{ij} y_j \geqslant dM_0 + (1-d)m_0$
- $M_1 = \max_i \sum_j P_{ij} y_j \leqslant dm_0 + (1-d)M_0$

Also ist $m_0 \leqslant m_1 \leqslant M_1 \leqslant M_0$.

Außerdem ist $M_1 - m_1 \leqslant (dm_0 + (1-d)M_0) - (dM_0 + (1-d)m_0) = (1-2d)(M_0 - m_0)$.

2. Durch Induktion ergibt sich hieraus für die kleinsten und größten Einträge m_k und M_k von $\mathbf{P}^k\mathbf{y}$: $m_0 \leqslant m_1 \leqslant \cdots m_k \leqslant M_k \leqslant \cdots \leqslant M_1 \leqslant M_0$ und $M_k - m_k \leqslant (1-2d)^k(M_0 - m_0)$.

   Die Folgen der m_k und der M_k sind also beschränkt und monoton, d. h. sie besitzen jeweils einen Grenzwert $m = \lim_{k\to\infty} m_k$ bzw. $M = \lim_{k\to\infty} M_k$.

3. O. B. d. A. nehmen wir nun an, dass $\mathbf{P}$ mindestens 2 Zeilen und Spalten hat (im Fall 1 ist die zu beweisende Aussage trivialerweise richtig). Folglich ist $0 < d \leqslant 1/2$ und damit $0 \leqslant 1 - 2d < 1$. Dann ist aber $\lim_{k\to\infty} M_k - m_k = 0$ und daher $M = m$.

4. Es sei $u = M = m$. Da alle Einträge in $\mathbf{P}^k\mathbf{y}$ zwischen m_k und M_k liegen, ist $\lim_{k\to\infty} \mathbf{P}^k\mathbf{y} = \mathbf{u}$, wobei $\mathbf{u}$ der konstante Vektor ist, dessen Einträge alle gleich u sind.

Wir betrachten nun den Fall, dass $\mathbf{y} = \mathbf{e}_j$ der Einheitsvektor ist, dessen j-te Komponente 1 ist und alle anderen gleich 0.

5. Dann ist $\mathbf{P}^k\mathbf{e}_j$ die j-te Spalte von $\mathbf{P}^k$. Für jedes j konvergiert also die Folge der j-ten Spalten von $\mathbf{P}^k$ gegen einen konstanten Vektor. Also existiert $\lim_{k\to\infty} \mathbf{P}^k = \mathbf{W}$ und besteht aus lauter konstanten Spalten, d. h. mit anderen Worten aus lauter gleichen Zeilen $\mathbf{w}$.

6. Um zu zeigen, dass alle Einträge in $\mathbf{w}$ echt größer 0 sind, benutzen wir die Voraussetzung, dass $\mathbf{P}$ keine Nulleinträge hat. Dann gilt für jedes j: $\mathbf{P}\mathbf{e}_j$ enthält nur echt positive Werte, d. h. in diesem Fall ist $m_1 > 0$ und daher auch $m > 0$. Dieses m ist aber gerade die j-te Komponente von $\mathbf{w}$.

7. Die Tatsache, dass $\sum_{i=1}^n w_i = 1$ ist, ergibt sich daraus, dass für alle k die Potenzen $\mathbf{P}^k$ stochastische Matrizen sind, d. h. Zeilensumme 1 haben.

■

Im Folgenden habe $\mathbf{w}$ stets die gleiche Bedeutung wie im obigen Beweis.

8.18 Satz. *Für jede ergodische Markov-Kette mit Matrix* $\mathbf{P}$ *gilt:*

1. $\mathbf{wP} = \mathbf{w}$.
2. *Falls* $\mathbf{vP} = \mathbf{v}$ *ist, ist* $\mathbf{v} = (\sum_j v_j)\mathbf{w}$.
3. *Es gibt genau eine Verteilung* $\mathbf{w}$ *(i. e. Summe der Einträge gleich* 1*) mit* $\mathbf{wP} = \mathbf{w}$.

8.19 Beweis.

1. $\mathbf{W} = \lim_{k\to\infty} \mathbf{P}^k = \lim_{k\to\infty} \mathbf{P}^{k+1} = (\lim_{k\to\infty} \mathbf{P}^k) \cdot \mathbf{P} = \mathbf{WP}$. Insbesondere gilt also für jede Zeile $\mathbf{w}$ von $\mathbf{W}$: $\mathbf{wP} = \mathbf{w}$.

2. Wenn $\mathbf{vP} = \mathbf{v}$ ist, dann auch $\mathbf{vP}^k = \mathbf{v}$ für jedes k und folglich $\mathbf{vW} = \mathbf{v}$. Ist $r = \sum_j v_j$ die Summe der Komponenten von $\mathbf{v}$, dann ist andererseits $\mathbf{vW} = r\mathbf{w}$, also $\mathbf{v} = r\mathbf{w}$.

3. Unter allen Vektoren $r\mathbf{w}$ gibt es offensichtlich genau einen, für den die Summe aller Einträge gleich 1 ist.

 © Th. Worsch 2000-2015

■

8.20 Definition Eine Verteilung $\mathbf{w}$ heißt *stationär*, falls $\mathbf{w} = \mathbf{wP}$ ist. ◇

8.21 Als erstes Beispiel wollen wir die stationäre Verteilung von Markov-Ketten M_G berechnen, die durch Graphen induziert werden. Dazu sei $G = (V, E)$ ein endlicher, zusammenhängender, ungerichteter Graphen, der nicht bipartit ist. Dann gehört jeder Knoten in G zu einem Zyklus ungerader Länge; außerdem gehört jeder Knoten zu einem Zyklus der Länge 2 (zu einem Nachbarn und zurück). Also hat in M_G jeder Zustand Periode 1 und die Markov-Kette ist aperiodisch. Da der Graph als ungerichtet und zusammenhängend angenommen wurde, ist die Kette auch irreduzibel, also auch ergodisch.

In diesem Fall kann man die eindeutige stationäre Verteilung $(w_1, \dots, w_n)$ leicht angeben:

8.22 Lemma. Für alle $v \in V$ ist $w_v = d(v)/2m$.

8.23 Beweis. Da die stationäre Verteilung gegebenenfalls eindeutig ist, genügt es nachzuweisen, dass $\mathbf{q}$ mit $q_v = d(v)/2m$ eine Verteilung und stationär ist.

$$\begin{aligned}
\sum_{v \in V} q_v &= \sum_{v \in V} d(v)/2m = 1/2m \sum_{v \in V} d(v) = 1 \, . \\
(\mathbf{qP})_v &= \sum_{u \in V} q_u P_{uv} = \sum_{(u,v) \in E} q_u P_{uv} = \sum_{(u,v) \in E} \frac{d(u)}{2m} \cdot \frac{1}{d(u)} = \sum_{(v,u) \in E} \frac{1}{2m} = \frac{d(v)}{2m} \, .
\end{aligned}$$

■

8.24 Wegen der Bemerkung in Punkt 8.15 gilt der dritte Teil der Aussage aus Satz 8.18 für irreduzible Markov-Ketten, auch wenn sie nicht aperiodisch sind: *Jede irreduzible Markov-Kette* $\mathbf{P}$ *besitzt genau eine stationäre Verteilung* $\mathbf{w}$.

Man beachte aber, dass $\lim_{t \to \infty} \mathbf{P}^t$ für irreduzible Markov-Ketten im allgemeinen nicht existiert! Für $\mathbf{P} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ und alle k ist z. B. $\mathbf{P}^{2k} = \mathbf{I}$ und $\mathbf{P}^{2k+1} = \mathbf{P}$.

8.25 Für ergodische Markov-Ketten existiert $\lim_{t \to \infty} \mathbf{P}^t = \mathbf{W}$. Folglich existiert auch der sogenannte Cesàro-Grenzwert $\lim_{t \to \infty} \mathbf{A}_t$, wobei $\mathbf{A}_t = \frac{1}{t+1} \sum_{k=0}^{t} \mathbf{P}^k$ ist und es ist $\lim_{t \to \infty} \mathbf{A}_t = \mathbf{W}$.

Da jeder Eintrag $P_{ij}^{(k)}$ die Wahrscheinlichkeit angibt, in k Schritten von i nach j zu gelangen, ist jeder Eintrag an Stelle (i, j) von $\mathbf{A}_t$ der erwartete Anteil (als Zahl zwischen 0 und 1) von Zeitpunkten zwischen 0 und t, zu denen man in Zustand j ist, wenn man in Zustand i startet. Diese Interpretation ist natürlich immer richtig (nicht nur für ergodische Markov-Ketten).

Der interessante Punkt ist, dass man auch für periodische irreduzible Markov-Ketten noch beweisen kann (wir unterlassen das hier):

8.26 Satz. *Es sei* $\mathbf{P}$ *die Übergangsmatrix einer irreduziblen Markov-Kette* M*. Dann gilt:*

- $\lim_{t \to \infty} \mathbf{A}_t = \mathbf{W}$ *existiert.*
- *Alle Zeilen von* $\mathbf{W}$ *sind gleich.*
- *Die Zeile* $\mathbf{w}$ *ist die eindeutig bestimmte stationäre Verteilung von* M*.*

Mit anderen Worten: Bezeichnet $N_i(j, t)$ *die Anzahl der Besuche von Zustand* j *während der ersten* t *Schritte bei Start in* i*, so ist* $\lim_{t \to \infty} N_i(j, t)/t = w_j$ *unabhängig vom Anfangszustand* i*.*

© Th. Worsch 2000-2015

Man mache sich an einem Gegenbeispiel klar, dass bei der folgenden Aussage die Forderung nach Aperiodizität unverzichtbar ist.

8.27 SATZ. *Für jede ergodische Markov-Kette* $\mathbf{P}$ *und jede Verteilung* $\mathbf{v}$ *gilt:* $\lim_{t\to\infty} \mathbf{v}\mathbf{P}^t = \mathbf{w}$.

8.28 BEWEIS. Es gilt $\lim_{k\to\infty} \mathbf{v}\mathbf{P}^k = \mathbf{v}\mathbf{W}$. Da sich die Einträge in $\mathbf{v}$ zu 1 summieren und alle Zeilen von $\mathbf{W}$ gleich $\mathbf{w}$ sind, ist $\mathbf{v}\mathbf{W} = \mathbf{w}$. ∎

8.29 SATZ. *Für jede irreduzible Markov-Kette mit stationärer Verteilung* $\mathbf{w} = (w_1, \dots, w_n)$ *gilt für alle* i*:* $w_i = 1/m_{ii}$.

8.30 BEWEIS. Wir beginnen mit einfachen Überlegungen zu den m_{ij}.

1. Falls $i \neq j$ ist, ist

$$m_{ij} = P_{ij} \cdot 1 + \sum_{k\neq j} P_{ik}(m_{kj} + 1) = 1 + \sum_{k\neq j} P_{ik} m_{kj}$$

2. Falls $i = j$ ist, ist

$$m_{ii} = P_{ii} \cdot 1 + \sum_{k\neq i} P_{ik}(m_{ki} + 1) = 1 + \sum_{k\neq i} P_{ik} m_{ki}$$

3. Es bezeichne nun $\mathbf{E}$ die Matrix, deren Einträge alle gleich 1 seien, $\mathbf{M}$ die Matrix mit

$$\mathbf{M}_{ij} = \begin{cases} m_{ij} & \text{falls } i \neq j \\ 0 & \text{falls } i = j \end{cases}$$

und $\mathbf{D}$ die Matrix mit

$$\mathbf{D}_{ij} = \begin{cases} 0 & \text{falls } i \neq j \\ m_{ii} & \text{falls } i = j \end{cases}$$

Dann lassen sich die eben genannten Gleichungen ausdrücken als Matrixgleichung

$$\mathbf{M} + \mathbf{D} = \mathbf{E} + \mathbf{P}\mathbf{M} \ .$$

Dazu äquivalent ist

$$(\mathbf{I} - \mathbf{P})\mathbf{M} = \mathbf{E} - \mathbf{D} \ .$$

Da $\mathbf{w}\mathbf{P} = \mathbf{w}$ ist, ist $\mathbf{w}(\mathbf{I} - \mathbf{P}) = \mathbf{0}$ und folglich $\mathbf{w}\mathbf{E} = \mathbf{w}\mathbf{D}$. Das bedeutet aber ausgeschrieben nichts anderes als

$$(1, 1, \dots, 1) = (w_1 m_{11}, w_2 m_{22}, \dots, w_n m_{nn})$$

∎

 © Th. Worsch 2000-2015

Zusammenfassung

1. Ergodische Markov-Ketten besitzen eine eindeutig bestimmte stationäre Verteilung.
2. Für irreduzible Markov-Ketten gilt das auch, aber man kann die stationäre Verteilung im allgemeinen nicht mehr durch Grenzwertübergang der P^k erhalten.

Literatur

Behrends, Ehrhard (2000). *Introduction to Markov Chains*. Advanced Lectures in Mathematics. Vieweg (siehe S. 69, 76).

Feller, W. (1968). *An Introduction to Probability Theory and Its Applications*. third. Bd. I. John Wiley & Sons (siehe S. 69).

Grinstead, Charles M. und J. Laurie Snell (1997). *Introduction to Probablity: Second Revised Edition*. American Mathematical Society. ISBN: 0-8218-0749-8 (siehe S. 69).

Kemeny, John G. und J. Laurie Snell (1976). *Finite Markov Chains*. Springer-Verlag (siehe S. 69).

 © Th. Worsch 2000-2015

9 Schnell mischende Markov-Ketten

Allgemeines zu schnell mischenden Markov-Ketten findet man zum Beispiel in dem Buch „*Introduction to Markov Chains*“ von Behrends (2000). Außerdem haben wir von einem Teil eines Vorlesungsskriptes von Steger (2001) über schnell mischende Markov-Ketten profitiert. Außerdem ist der Überblick von Randall (2006) empfehlenswert.

9.1 Der Metropolis-Algorithmus

9.1 Definition Eine ergodische Markov-Kette heißt *(zeit-)reversibel* (im Englischen mitunter auch *in detailed balance*), wenn für die stationäre Verteilung $\mathbf{w}$ und alle Zustände i und j gilt:

$$\mathbf{w}_i P_{ij} = \mathbf{w}_j P_{ji} \,. \qquad \diamond$$

Zur Begründung dieses Namens mache man sich klar, dass $\mathbf{w}_i P_{ij}$ die Wahrscheinlichkeit ist, dass man bei einer Markov-Kette im stationären Zustand bei einem Schritt gerade einen Übergang von Zustand i nach Zustand j beobachtet, und $\mathbf{w}_j P_{ji}$ die Wahrscheinlichkeit, dass man bei einem Schritt umgekehrt einen Übergang von Zustand j nach Zustand i.

Als Beispiele können die folgenden Ketten dienen:

9.2 Definition Es sei $G = (V, E)$ ein zusammenhängender ungerichteter Graph ohne Schlingen und $0 < \beta \leqslant 1$ eine reelle Zahl. Mit $d(i)$ bezeichnen wir den Grad von Knoten i und es sei $d = \max_{i \in V} d(i)$. Die Übergangswahrscheinlichkeiten P_{ij} der Markov-Kette $M_{G,\beta}$ sind dann wie folgt definiert:

$$P_{ij} = \begin{cases} \beta/d & \text{falls } i \neq j \text{ und } (i,j) \in E \\ 0 & \text{falls } i \neq j \text{ und } (i,j) \notin E \\ 1 - d(i)\beta/d & \text{falls } i = j \end{cases} \qquad \diamond$$

9.3 Man kann sich überlegen, dass für die so definierten Markov-Ketten die Gleichverteilung die stationäre Verteilung ist. Da der Graph zusammenhängend ist, ist die Kette irreduzibel. Für $\beta < 1$ ist sie außerdem aperiodisch und reversibel.

9.4 Eine kleine Verallgemeinerung liefert schon eine ganz einfache Variante des sogenannten *Metropolis-Hastings-Algorithmus*. Sei dazu $\mathbf{p}$ eine Wahrscheinlichkeitsverteilung auf V, die nirgends 0 ist. Dann betrachte man folgende Festlegungen:

$$P_{ij} = \begin{cases} \min(1, \frac{\mathbf{p}_j}{\mathbf{p}_i}) \cdot \beta/d & \text{falls } i \neq j \text{ und } (i,j) \in E \\ 0 & \text{falls } i \neq j \text{ und } (i,j) \notin E \\ 1 - \sum_{i \neq k} P_{ik} & \text{falls } i = j \end{cases}$$

Auch diese Markov-Kette ist reversibel, was man zum Beispiel so sieht. Sei $i \neq j$ und $(i,j) \in E$ (die anderen Fälle sind trivial). Dann gilt:

$$\frac{\mathbf{p}_i P_{ij}}{\mathbf{p}_j P_{ji}} = \frac{\mathbf{p}_i \min(1, \frac{\mathbf{p}_j}{\mathbf{p}_i}) \cdot \beta/d}{\mathbf{p}_j \min(1, \frac{\mathbf{p}_i}{\mathbf{p}_j}) \cdot \beta/d} = \begin{cases} \frac{\mathbf{p}_i}{\mathbf{p}_j} / \frac{\mathbf{p}_i}{\mathbf{p}_j} & \text{falls } \mathbf{p}_i \leqslant \mathbf{p}_j \\ \frac{\mathbf{p}_i}{\mathbf{p}_j} \cdot \frac{\mathbf{p}_j}{\mathbf{p}_i} & \text{falls } \mathbf{p}_i > \mathbf{p}_j \end{cases} = 1$$

Wegen des folgenden Lemmas kennt man auch sofort die stationäre Verteilung dieser Markov-Kette: Es ist $\mathbf{p}$.

9.5 Lemma. Ist M eine ergodische Markov-Kette und $\mathbf{q}$ eine Verteilung mit der Eigenschaft, dass für alle Zustände i und j gilt, dass $\mathbf{q}_i P_{ij} = \mathbf{q}_j P_{ji}$ ist, dann ist $\mathbf{q}$ die (eindeutig bestimmte) stationäre Verteilung von M.

9.6 Beweis. Für alle i ist

$$(\mathbf{q}P)_i = \sum_j \mathbf{q}_j P_{ji} = \sum_j \mathbf{q}_i P_{ij} = \mathbf{q}_i \sum_j P_{ij} = \mathbf{q}_i \ .$$

■

9.7 Für den klassischen Algorithmus von Metropolis u. a. (1953) ist eine zeilenstochastische Matrix Q gegeben, die symmetrisch ist (diese Einschränkung wurde von Hastings (1970) beseitigt). Ziel ist aber eine reversible Markov-Kette mit Übergangsmatrix P, deren stationäre Verteilung $\mathbf{w}$ nur indirekt gegeben ist. Festgelegt ist nämlich nur eine Funktion $H\colon S \to \mathbb{R}$ mit der Eigenschaft, dass $e^{-H(i)}$ proportional zur Wahrscheinlichkeit von Zustand i in $\mathbf{w}$ ist. Der Proportionalitätsfaktor ist natürlich $Z = \sum_{i \in S} e^{-H(i)}$.

Q heißt üblicherweise *proposal matrix*, H *energy function* und Z *partition function*.

Das Schöne am Metropolis-Algorithmus ist, dass man Z gar nicht kennen muss. (In manchen Anwendungen in der Physik ist S *sehr* groß.)

9.8 Algorithmus. Der Metropolis-Algorithmus, oder auch *Metropolis sampler*, funktioniert wie folgt:

Man startet in einem beliebigen Zustand $i_0 \in S$. Ist man nach t Schritten in Zustand $i_t \in S$, dann ergibt sich i_{t+1} in zwei Teilschritten:

- Zunächst wird gemäß der Verteilung $Q(i_t, \cdot)$ zufällig ein $j \in S$ bestimmt.
- Dann gibt es zwei Möglichkeiten:
  - Falls $H(j) \leqslant H(i_t)$ ist, also $w_j \geqslant w_{i_t}$, dann wählt man $i_{t+1} = j$.
  - Falls $H(j) > H(i_t)$ ist, also $w_j < w_{i_t}$, wählt man

$$i_{t+1} = \begin{cases} j & \text{mit Wahrscheinlichkeit } w_j / w_{i_t} = e^{H(i_t) - H(j)} \\ i_t & \text{sonst} \end{cases}$$

9.9 Satz. *Die durch den Metropolis-Algorithmus festgelegte Markov-Kette auf S ist reversibel und in ihrer stationären Verteilung hat Zustand i Wahrscheinlichkeit $e^{-H(i)}/Z$.*

9.10 Beweis. Sei P die durch den Algorithmus festgelegte Übergangsmatrix. Es genügt zu beweisen, dass für alle $i, j \in S$ Balancierheit vorliegt, d. h.:

$$e^{-H(i)}/Z \cdot P_{ij} = e^{-H(j)}/Z \cdot P_{ji}$$

Das ist eine einfache Rechnung analog zu der in Punkt 9.4. ■

 © Th. Worsch 2000-2015

9.2 Anmerkungen zu Eigenwerten

9.11 Lemma. Es sei P eine zeilenstochastische Matrix. Dann ist 1 ein Eigenwert von P und für jeden Eigenwert λ von P gilt: $|\lambda| \leqslant 1$.

9.12 Beweis. Der erste Teil der Aussage ist wegen $P(1,\dots,1)^\top = (1,\dots,1)^\top$ klar.

Sei nun $\lambda \in \mathbb{C}$ Eigenwert und $P(x_1,\dots,x_n)^\top = \lambda(x_1,\dots,x_n)^\top$. Es sei i_0 ein Index mit $|x_{i_0}| = \max_j |x_j|$. Es liegen also alle x_j im Kreis um den Ursprung mit Radius $|x_{i_0}|$. Es gilt:

$$|\lambda| \cdot |x_{i_0}| = |\lambda \cdot x_{i_0}| = \left|\sum_j P_{i_0 j} x_j\right| \leqslant \sum_j P_{i_0 j}|x_j| \leqslant \sum_j P_{i_0 j}|x_{i_0}| = |x_{i_0}| \sum_j P_{i_0 j} = |x_{i_0}| \,.$$

Also ist $|\lambda| \leqslant 1$. ∎

Für reversible Markov-Ketten kann man die Aussage von Lemma 9.11 verschärfen.

9.13 Lemma. Ist P die stochastische Matrix einer reversiblen Markov-Kette, dann hat P nur reelle Eigenwerte.

Für einen Beweis konsultiere man z. B. das Vorlesungsskript von Steger (2001) oder Kapitel 3 im Manuskript von Aldous und Fill (1999)

9.14 Im folgenden benutzen wir die Abkürzung $\lambda_{max} = \max\{\, |\lambda| \mid \lambda \text{ ist Eigenwert von } P \text{ und } \lambda \neq 1\}$. Auf Grund des Vorangegangenen ist klar, dass $\lambda_{max} \leqslant 1$ ist.

9.15 Falls $\lambda_N < 0$ ist geht man auf bewährte Weise von $\mathbf{P}$ zur Matrix $\mathbf{P}' = \frac{1}{2}(\mathbf{P} + \mathbf{I})$ über. Für deren Eigenwerte gilt dann $\lambda_i' = \frac{1}{2}(\lambda_i + 1)$, so dass alle Eigenwerte echt größer 0 sind. In diesem Fall ist dann also $\lambda_{max} = \lambda_2$.

9.3 Schnell mischende Markov-Ketten

Wir interessieren uns nun für die Frage, wie lange es dauert, bis man bei einer Markov-Kette, die man mit einer Verteilung oder in einem bestimmten Zustand begonnen hat, davon ausgehen kann, dass die Wahrscheinlichkeiten, in bestimmten Zuständen zu sein, denen der stationären Verteilung „sehr nahe“ sind.

Dazu definieren wir als erstes eine Art Abstandsbegriff für diskrete Wahrscheinlichkeitsverteilungen.

9.16 Definition Für zwei Verteilungen $\mathbf{p}$ und $\mathbf{q}$ ist die *totale Variationsdistanz*

$$\|p - q\|_{tv} = \frac{1}{2} \sum_{j \in S} |\mathbf{p}_j - \mathbf{q}_j| \,.$$

◇

Man kann sich überlegen, dass $\|p - q\|_{tv} = \max_{T \subseteq S} |\mathbf{p}(T) - \mathbf{q}(T)|$ ist, wobei $\mathbf{p}(T)$ zu verstehen ist als $\sum_{j \in T} \mathbf{p}_j$ (und analog $\mathbf{q}(T)$).

© Th. Worsch 2000-2015

9.17 Definition Für eine ergodische Markov-Kette mit Matrix $\mathbf{P}$ und stationärer Verteilung $\mathbf{w}$ und für alle Verteilungen $\mathbf{p}$ sei

$$\delta_{\mathbf{p}}(t) = \|\mathbf{p}\mathbf{P}^t - w\|_{tv}$$

die totale Variationsdistanz zwischen $\mathbf{w}$ und der Verteilung, die man nach t Schritten ausgehend von $\mathbf{p}$ erreicht hat.

Als *Variationsdistanz zum Zeitpunkt* t bezeichnen wir das Maximum $\Delta(t) = \max_{\mathbf{p}} \delta_{\mathbf{p}}(t)$. ◇

Man kann zeigen, dass das Maximum für einen Einheitsvektor $\mathbf{e}_i = (0, \dots, 0, 1, 0, \dots, 0)$ angenommen wird. Also gilt:

$$\Delta(t) = \max_i \|P_i^t - w\|_{tv}$$

wobei $\mathbf{P}_i^t$ die i-te Zeile von $\mathbf{P}^t$ sei.

9.18 Im folgenden benutzen wir die Abkürzung

$$w_{min} = \min_j \mathbf{w}_j$$

9.19 Satz. *Für jede reversible Markov-Kette mit stationärer Verteilung* $\mathbf{w}$ *gilt:*

$$\Delta(t) \leqslant \frac{\lambda_{max}^t}{w_{min}} .$$

Wenn also $\lambda_{max} < 1$ ist (was bei reversiblen Markov-Ketten der Fall ist), dann nähert sich die Markov-Kette in einem gewissen Sinne „schnell“ der stationären Verteilung. Wir präzisieren das noch wie folgt:

9.20 Häufig ist man an einer ganzen Familie von Markov-Ketten $M(I)$ interessiert. Dabei ergibt sich jedes $M(I)$ auf Grund einer Instanz I des eigentlich zu lösenden Problems. Zum Beispiel könnte jedes I ein Graph sein, und die Zustände von $M(I)$ sind gerade die Matchings von I.

9.21 Definition Es sei M eine ergodische Markov-Kette mit stationärer Verteilung $\mathbf{w}$. Für $\varepsilon > 0$ sei

$$\tau(\varepsilon) = \min\{t \mid \forall t' \geqslant t : \Delta(t') \leqslant \varepsilon\}$$

die ε-*Konvergenzzeit* der Markov-Kette M. ◇

9.22 Definition Eine Familie von Markov-Ketten $M(I)$ heißt *schnell mischend*, falls die ε-Konvergenzzeit polynomiell in $|I|$ und $\ln 1/\varepsilon$ ist. ◇

9.23 Wegen Satz 9.19 ist eine reversible Markov-Kette jedenfalls dann schnell mischend, wenn für ein t, das polynomiell in $|I|$ und $\log 1/\varepsilon$ ist, gilt:

$$\frac{\lambda_{max}^t}{w_{min}} \leqslant \varepsilon .$$

Äquivalente Umformungen ergeben

$$\begin{aligned} \lambda_{max}^t &\leqslant \varepsilon w_{min} \\ \left(\frac{1}{\lambda_{max}}\right)^t &\geqslant \frac{1}{\varepsilon w_{min}} \\ t &\geqslant \frac{\ln \varepsilon^{-1} + \ln w_{min}^{-1}}{\ln \lambda_{max}^{-1}} \end{aligned}$$

 © Th. Worsch 2000-2015

Wegen $1 - x \leqslant \ln x^{-1}$ für $0 < x < 1$ ist das jedenfalls dann der Fall, wenn

$$t \geqslant \frac{\ln \varepsilon^{-1} + \ln w_{min}^{-1}}{1 - \lambda_{max}}$$

Schnelles Mischen liegt also jedenfalls dann vor, wenn $\ln w_{min}^{-1}$ und $1/(1-\lambda_{max})$ polynomiell in $|I|$ sind.

Damit haben wir zumindest eine Hälfte des folgenden Satzes bewiesen, der obere und untere Schranken für $\tau(\varepsilon)$ angibt:

9.24 Satz.

$$\tau(\varepsilon) \leqslant \frac{1}{1-\lambda_{max}} \log \frac{1}{w_{min}\varepsilon}$$
$$\tau(\varepsilon) \geqslant \frac{1}{2(1-\lambda_{max})} \log \frac{1}{2\varepsilon}$$

Es stellt sich die Frage, woher man (zumindest näherungsweise) Kenntnis von λ_{max} bzw. im Falle von reversiblen Markov-Ketten von λ_2 bekommen kann. Eine Möglichkeit ist der sogenannte Leitwert einer Markov-Kette. Wir werden ihn mit Hilfe gewichteter Graphen einführen, die später selbst noch nützlich sein werden.

9.25 Definition Für eine reversible Markov-Kette $M = (S, \mathbf{P})$ mit stationärer Verteilung $\mathbf{w}$ sei F_M der gerichtete gewichtete Graph mit Knotenmenge S und Kantenmenge $E_F = \{(i,j) \mid i \neq j \wedge P_{ij} > 0\}$. Jede Kante (i,j) ist gewichtet mit der reellen Zahl $c(i,j) = \mathbf{w}_i P_{ij}$. ◇

9.26 Definition Für eine reversible Markov-Kette mit Zustandsmenge S und stationärer Verteilung $\mathbf{w}$ definieren wir für jede Teilmenge $T \subseteq S$

$$\begin{array}{lrcl} \text{die Kapazität} & C(T) & = & \sum_{i \in T} \mathbf{w}_i \\ \text{den Fluß} & F(T) & = & \sum_{i \in T, j \notin T} \mathbf{w}_i \mathbf{P}_{ij} \\ \text{und} & \Phi(T) & = & F(T)/C(T) \end{array}$$

Der *Leitwert* Φ der Markov-Kette ist dann

$$\Phi = \min_{T \subseteq S} \max\{\Phi(T), \Phi(S \setminus T)\} .$$

◇

Eine kurze Überlegung zeigt, dass $\Phi(T)$ die bedingte Wahrscheinlichkeit ist, dass man bei der Markov-Kette mit stationärer Verteilung einen Übergang von innerhalb von T nach außerhalb von T beobachtet. Wenn $\Phi(T)$ klein ist, dann ist T sozusagen eine Art „Falle“, aus der die Markov-Kette schlecht heraus kommt.

Man kann nun mit einigem technischen Aufwand zeigen:

9.27 Satz. *Für jede reversible Markov-Kette gilt:*

$$1 - 2\Phi^2 \leqslant \lambda_2 \leqslant 1 - \frac{\Phi^2}{2} .$$

Zusammen mit Punkt 9.23 ergibt sich:

 © Th. Worsch 2000-2015

9.28 KOROLLAR. Für reversible Markov-Ketten (mit $\lambda_2 = \lambda_{max}$) ist

$$\tau(\varepsilon) \leqslant \frac{2}{\Phi^2}(\ln \varepsilon^{-1} + \ln w_{min}^{-1})$$

Man kennt mehrere Methoden, um den Leitwert jedenfalls mancher Markov-Ketten zu berechnen.

Die folgende Definition ist eine Art graphentheoretische Version des Leitwertes:

9.29 DEFINITION Für einen ungerichteten Graphen (V, E) ist die *Kantenvervielfachung* μ das Minimum der Zahlen

$$\frac{|\{(i,j) \mid i \in T \wedge j \notin T \wedge (i,j) \in E\}|}{|T|}$$

wobei über alle Teilmengen $T \subseteq V$ minimiert werde mit $|T| \leqslant |V|/2$. ◇

9.30 Man kann sich überlegen, dass für die Markov-Ketten $M_{G,\beta}$ aus Definition 9.2 gilt: $\Phi = \beta\mu/d$.

Damit ist man bei der Aufgabe gelandet, die Kantenvervielfachung von Graphen zu bestimmen. Das kann man zum Beispiel mit Hilfe der Methode der sogenannten kanonischen Pfade von Sinclair machen. Die Verallgemeinerung für beliebige reversible Markov-Ketten betrachtet Mehrgüterflüsse.

9.31 DEFINITION Für jedes Paar (i, j) von Knoten in F_M soll von einem „Gut" g_{ij} die Menge $\mathbf{w}_i\mathbf{w}_j$ von i nach j transportiert werden. Dazu werden Flüsse $f_{ij} : E_F \to \mathbb{R}_+$ gesucht, so dass die folgenden naheliegenden Forderungen erfüllt sind:

$$\begin{aligned} \sum_k f_{ij}(i,k) &= \mathbf{w}_i\mathbf{w}_j \\ \text{für alle } l \neq i,j: \sum_k f_{ij}(k,l) &= \sum_m f_{ij}(l,m) \\ \sum_k f_{ij}(k,j) &= \mathbf{w}_i\mathbf{w}_j \end{aligned}$$

Der Gesamtfluss durch eine Kante e sei

$$f(e) = \sum_{i \neq j} f_{ij}(e)$$

und die *relative Kantenauslastung*

$$\rho(f) = \max_{e \in E_F} f(e)/c(e) \; .$$

◇

Dann gilt die folgende Aussage, die wir hier nicht beweisen:

9.32 LEMMA. Für jede Markov-Kette mit Flüssen f_{ij} gilt:

$$\Phi \geqslant \frac{1}{2\rho(f)} \; .$$

Um auf einen großen Leitwert schließen zu können, muss man daher versuchen, Flüsse mit kleiner (i. e. polynomieller) relativer Kantenauslastung zu finden.

Wir wollen dies nun auf Random Walks im Hyperwürfel anwenden.

 © Th. Worsch 2000-2015

9.33 BEISPIEL. Dazu sei eine Dimensionalität n beliebig aber fest gewählt und M die Markov-Kette, die sich gemäß Definition 9.2 für $\beta = 1/2$ aus dem n-dimensionalen Hyperwürfel H_n als zu Grunde liegenden Graphen ergibt. Das n sei per definitionem die „Größe“ der Probleminstanz.

M ist reversibel gemäß Punkt 9.3. Da in H_n jeder Knoten Grad n hat, sind die Übergangswahrscheinlichkeiten also $P_{ii} = 1/2$ und $P_{ij} = 1/2n$ für $i \neq j$. Aus Symmetriegründen ist klar, dass die stationäre Verteilung die Gleichverteilung ist mit $\mathbf{w}_i = 1/2^n$; damit ist natürlich auch $w_{min} = 1/2^n$.

Offensichtlich ist $\ln 1/w_{min} \in \Theta(n)$ polynomiell in n. Um einzusehen, dass M schnell mischend ist, genügt es folglich wegen Lemma 9.32, Flüsse f_{ij} zu finden, so dass $\rho(f)$ polynomiell (in n) ist.

Dazu gehen wir wie folgt vor. Jeder Fluss f_{ij} muss gerade die „Menge“ $1/2^{2n}$ transportieren. Sie wird wie folgt verteilt: Zwischen i und j gibt es $d!$ kürzeste Pfade, wobei d die Hammingdistanz zwischen i und j ist. Auf jedem dieser Pfade transportieren wir den gleichen Anteil der Gesamtmenge.

Die Bestimmung der relativen Kantenauslastung wird dadurch erleichtert, dass aus Symmetriegründen auf jeder Kante der gleiche Gesamtfluss vorliegt.

Für jedes d gibt es $2^n \cdot \binom{n}{d}$ Paare (i, j) mit Abstand d. Für ein *festes* Paar (i, j) haben alle für den Fluss f_{ij} verwendeten Pfade Länge d und es ist folglich

$$\sum_{e \in E_F} f_{ij}(e) = d\mathbf{w}_i\mathbf{w}_j \ .$$

Also ist

$$\begin{aligned}
f(e) &= \frac{1}{|E_F|} \sum_{d=1}^{n} 2^n \binom{n}{d} \cdot d \cdot \mathbf{w}_i\mathbf{w}_j \\
&= \frac{1}{n2^n} \sum_{d=1}^{n} 2^n \binom{n}{d} \cdot d \cdot \frac{1}{2^{2n}} \\
&= \frac{1}{2^{2n}} \sum_{d=1}^{n} \binom{n}{d} \cdot \frac{d}{n} \\
&= \frac{1}{2^{2n}} \sum_{d=1}^{n} \binom{n-1}{d-1} \\
&= \frac{1}{2^{2n}} \sum_{d=0}^{n-1} \binom{n-1}{d} \\
&= \frac{2^{n-1}}{2^{2n}} = \frac{1}{2 \cdot 2^n}
\end{aligned}$$

Andererseits ist für alle Kanten $c(e) = \mathbf{w}_i P_{ij} = 1/(2n \cdot 2^n)$ und somit

$$\rho(f) = \frac{1/(2 \cdot 2^n)}{1/(2n \cdot 2^n)} = n \ .$$

Wegen Korollar 9.28 sind diese Markov-Ketten also schnell mischend.

Man mache sich noch einmal klar, was das bedeutet (siehe Definition 9.22): Es sei $\mathbf{p} = \mathbf{e}_0$ die Anfangs-„Verteilung“ bei der man sicher im Knoten $(0, 0, \dots, 0)$ des n-dimensionalen Hyperwürfels startet. Es sei $\mathbf{p}_t$ die nach t Schritten erreichte Wahrscheinlichkeitsverteilung. Für jedes $\varepsilon = 2^{-k}$ ist dann die ε-Konvergenzzeit, also die Anzahl Schritte nach der $\|p_t - w\|_{tv} < \varepsilon$ immer gilt, polynomiell in $\ln 1/\varepsilon = k$ und n. Und das, obwohl der Hyperwürfel 2^n Knoten hat!

 © Th. Worsch 2000-2015

Zusammenfassung

Oft hat man es mit ergodischen Markov-Ketten zu tun, die sogar reversibel sind. In diesem Fall gibt es Kriterien, um festzustellen, ob sie schnell mischend sind.

Literatur

Aldous, David und James Allen Fill (1999). „Reversible Markov Chains and Random Walks on Graphs“. In: URL: http://www.stat.berkeley.edu/~aldous/RWG/book.html (siehe S. 78).

Behrends, Ehrhard (2000). *Introduction to Markov Chains*. Advanced Lectures in Mathematics. Vieweg (siehe S. 69, 76).

Hastings (1970). „Monte Carlo Sampling Methods Using Markov Chains and Their Applications“. In: *Biometrika* 57.1, S. 97–109. DOI: 10.1093/biomet/57.1.97 (siehe S. 77).

Metropolis, Nicholas u. a. (1953). „Equations of State Calculations by Fast Computing Machines“. In: *Journal of Chemical Physics* 21.6, S. 1087–1092 (siehe S. 77).

Randall, Dana (2006). „Rapidly Mixing Markov Chains with Applications in Computer Science and Physics“. In: *Computing in Science and Engineering* 8.2, S. 30–41 (siehe S. 76).

Steger, Angelika (2001). „Schnell mischende Markov-Ketten“. In: URL: http://wwwmayr.informatik.tu-muenchen.de/lehre/2001SS/ra/slides/markov-skript.ps (siehe S. 76, 78).

 © Th. Worsch 2000-2015

10 Randomisiertes Approximieren

Im ersten Abschnitt dieses Kapitels führen wir Begriffe für verschiedene Varianten sogenannter polynomieller Approximationsschemata ein. Im zweiten Abschnitt stellen wir randomisierte polynomielle Approximationsschemata für das Problem vor, die Anzahl erfüllender Belegungen der Variablen einer Formel in DNF zu bestimmen.

10.1 Polynomielle Approximationsschemata

10.1 Im folgenden sei stets Π ein „Zählproblem“, das für jede Eingabe x eine gewisse Anzahl $\#_\Pi(x)$ (oder kurz $\#(x)$) von „Lösungen“ besitzt.

10.2 Jedem solchen Zählproblem Π entspricht ein Entscheidungsproblem E_Π, bei dem es um die Frage geht, ob es für eine Eingabe x eine Lösung gibt, d. h. ob $\#(x) > 0$ ist.

10.3 Definition Ein Problem Π gehört zur Klasse **#P**, wenn es eine nichtdeterministische Turingmaschine gibt, die in Polynomialzeit arbeitet und für jede Eingabe x genau $\#_\Pi(x)$ akzeptierende Berechnungen besitzt.

Ein Problem Π ist **#P**-*vollständig*, wenn jedes Problem $\Pi' \in$ **#P** von einer Turingmaschine in Polynomialzeit auf Π reduziert werden kann. ◇

Unter den **#P**-vollständigen Problemen finden sich unter anderem:

- Wieviele erfüllende Belegungen hat eine Formel, die in DNF gegeben ist?
- Wieviele erfüllende Belegungen hat eine 2sat-Formel?
- Wieviele perfekte Matchings hat ein gegebener bipartiter Graph?
- Was ist die Permanente einer gegebenen Booleschen Matrix?
- Wieviele topologische Sortierungen eines gegebenen DAG gibt es?

Dabei ist es insbesondere erstaunlich, dass die zugehörigen Entscheidungsvarianten der beiden ersten Probleme sehr einfach sind.

10.4 Lemma. Kann man ein **#P**-vollständiges Problem deterministisch in Polynomialzeit lösen, dann ist **P** = **NP**.

Es ist daher naheliegend, sich für Algorithmen zu interessieren, die Zählprobleme jedenfalls näherungsweise lösen:

10.5 Definition Ein *Approximationsschema (AS)* für Π ist ein deterministischer Algorithmus A, der für jede Eingabe x der Größe $n = |x|$ und für jedes $\varepsilon > 0$ eine Ausgabe $A(x)$ erzeugt, für die gilt:

$$(1-\varepsilon)\#(x) \leqslant A(x) \leqslant (1+\varepsilon)\#(x) .$$

$A(x)$ heißt dann eine ε-*Approximation* von $\#(x)$.

Ein *polynomielles Approximationsschema (PAS)* ist ein AS, dessen Laufzeit polynomiell in n ist. Ein *voll polynomielles Approximationsschema (FPAS)* ist ein AS, dessen Laufzeit polynomiell in n und $1/\varepsilon$ ist. ◇

Für **#P**-vollständige Probleme kennt man keine PAS oder gar FPAS. Die Situation wird besser, wenn man Approximationsschemata mit Zufallsentscheidungen erlaubt:

10.6 Definition Ein *randomisiertes Approximationsschema (RAS)* für Π ist ein randomisierter Algorithmus A, der für jede Eingabe x der Größe $n = |x|$ und für jedes $\varepsilon > 0$ eine Ausgabe $A(x)$ erzeugt, für die gilt:

$$\mathbf{Pr}\left[(1-\varepsilon)\#(x) \leqslant A(x) \leqslant (1+\varepsilon)\#(x)\right] \geqslant \frac{3}{4} .$$

Ein *polynomielles randomisiertes Approximationsschema (PRAS)* ist ein RAS, dessen Laufzeit polynomiell in n ist. Ein *voll polynomielles randomisiertes Approximationsschema (FPRAS)* ist ein RAS, dessen Laufzeit polynomiell in n und $1/\varepsilon$ ist. ◇

Die Wahl der Wahrscheinlichkeit 3/4 in Definition 10.6 ist bis zu einem gewissen Grad willkürlich. Man überlege sich als Übungsaufgabe, dass jede Konstante echt größer 1/2 „genauso gut" ist.

10.7 Definition Ein (ε, δ)-*FPRAS* ist ein FPRAS, das für jede Eingabe x mit einer Wahrscheinlichkeit größer oder gleich $1-\delta$ eine ε-Approximation berechnet und dessen Laufzeit polynomiell in n, $1/\varepsilon$ und $\log 1/\delta$ ist. ◇

10.8 Ein PRAS für ein Zählproblem Π kann man in einen randomisierten Algorithmus für das Entscheidungsproblem umwandeln, der zeigt, dass E_Π in **BPP** liegt.

Hätte man ein PRAS für ein **NP**-vollständiges Problem, so wäre **NP** $\subseteq$ **BPP**. Experten erwarten nicht, dass letzteres der Fall ist, und sie erwarten daher auch nicht, dass man ein PRAS für ein **NP**-vollständiges Problem findet.

10.2 Zählen von Lösungen von Formeln in DNF

10.9 Problem. Es sei $F(x_1, \ldots, x_n) = C_1 \vee \cdots \vee C_m$ mit $C_1 = l_{1,1} \wedge \cdots \wedge, l_{1,r_1}, \ldots, C_m = l_{m,1} \wedge \cdots \wedge, l_{m,r_m}$ eine Formel in DNF, so dass jedes Literal $l_{i,j}$ eine Variable x_k oder ihre Negation $\overline{x_k}$ ist und jede Variable in jeder Klausel C_i höchstens einmal (sei es normal oder negiert) vorkommt. Es sei $n \geqslant 1$ und $m \geqslant 1$.

Es bezeichne $\#F$ die Anzahl der F erfüllenden Variablenbelegungen $\mathbf{x} : \{x_1, \ldots, x_n\} \to \{0, 1\}$. Die zu lösende Aufgabe besteht darin, zu gegebenem F die Zahl $\#F$ zu bestimmen.

10.10 Man weiß, dass dieses Problem **#P**-vollständig ist. Die Existenz eines deterministischen Algorithmus dafür, der in Polynomialzeit arbeitet, wird daher (analog wie bei **NP**) von vielen bezweifelt.

Wir werden im folgenden aber ein (ε, δ)-FPRAS vorstellen und analysieren.

Dazu gehen wir zunächst zu der folgenden allgemeineren Problemstellung über und untersuchen einen ersten einfachen Algorithmus.

10.11 Es sei U ein endliches „Universum" und $G \subseteq U$ eine Teilmenge, deren Größe bestimmt werden soll. G sei gegeben mittels der charakteristische Funktion $g : U \to \{0, 1\}$ von G; es sei also $g(u) = 1 \iff u \in G$. Gelegentlich benutzen wir die Abkürzung $\rho = |G| / |U|$.

Wir machen folgende Annahmen:

- g ist schnell zu berechnen, d. h. die Frage der Zugehörigkeit eines u zu G ist schnell zu entscheiden.

 © Th. Worsch 2000-2015

- Man kann schnell zufällig gleichverteilt ein u aus U auswählen.

Im Falle der Bestimmung von $\#F$ ist U die Menge aller 2^n möglichen Variablenbelegungen $\mathbf{x}$ und die Abbildung g ist gerade die durch die Auswertung $F(\mathbf{x})$ vermittelte.

10.12 ALGORITHMUS.

$z \leftarrow 0$ ⟨*Variable zum Zählen der Lösungen*⟩
for $i \leftarrow 1$ **to** N **do**
  $u \leftarrow$ ⟨*zufällig gleichverteilt aus* U *ausgewähltes Element*⟩
  if $g(u) = 1$ **then**
    $z \leftarrow z + 1$
  fi
od
return $\frac{z}{N} \cdot |U|$

Wir gehen zunächst kurz darauf ein, dass dieser Algorithmus sinnvoll ist. Anschließend wird aber plausibel gemacht werden, dass er leider noch einen Nachteil hat.

10.13 LEMMA. Es sei Y_i die Zufallsvariable mit

$$Y_i = \begin{cases} 1 & \text{falls im } i\text{-ten Schleifendurchlauf } g(u) = 1 \text{ ist} \\ 0 & \text{sonst} \end{cases}$$

und $Z = \frac{|U|}{N} \sum_{i=1}^{N} Y_i$. Dann ist $\mathbf{E}[Z] = |G|$.

10.14 BEWEIS. Offensichtlich ist $\mathbf{E}[Y_i] = |G| / |U|$. Wegen der Linearität des Erwartungswertes ist $\mathbf{E}[Z] = \frac{|U|}{N} \sum_{i=1}^{N} \mathbf{E}[Y_i] = \frac{|U|}{N} \cdot N \cdot \frac{|G|}{|U|} = |G|$. ∎

Durch dieses Lemma noch nicht beantwortet ist aber die Frage, wie groß man die Anzahl N der Schleifendurchläufe wählen sollte, damit die Wahrscheinlichkeit für ein „stark“ von $|G|$ abweichendes Ergebnis „klein“ ist.

Darauf gibt der folgende Satz einen Hinweis:

10.15 SATZ. *Es seien δ und ε aus dem Intervall $(0;1]$. Algorithmus 10.12 liefert mit einer Wahrscheinlichkeit größer oder gleich $1-\delta$ eine ε-Approximation für $|G|$, falls gilt:*

$$N \geqslant \frac{5}{\varepsilon^2 \rho} \ln \frac{2}{\delta} .$$

10.16 BEWEIS. Es seien δ und ε beliebig aber fest. Es sei $Y = \sum_{i=1}^{N} Y_i$ und folglich $Z = |U| Y/N$.

Y ist binomialverteilt mit Erwartungswert $N\rho$ und auf Y sind die Ergebnisse über Chernoff-Schranken anwendbar, z. B. Korollar 4.21. Damit erhält man:

$$\begin{aligned} & \mathbf{Pr}[(1-\varepsilon)|G| \leqslant Z \leqslant (1+\varepsilon)|G|] \\ = \; & \mathbf{Pr}[(1-\varepsilon)N\rho \leqslant Y \leqslant (1+\varepsilon)N\rho] \\ = \; & 1 - \mathbf{Pr}[(1-\varepsilon)N\rho > Y] - \mathbf{Pr}[Y > (1+\varepsilon)N\rho] \\ \geqslant \; & 1 - e^{-\varepsilon^2 N\rho/2} - e^{-\varepsilon^2 N\rho/5} \\ \geqslant \; & 1 - 2e^{-\varepsilon^2 N\rho/5} . \end{aligned}$$

Setzt man hier den oben angegebenen Wert für N ein, ergibt sich im Exponenten

$$-\frac{\varepsilon^2 N\rho}{5} = -\frac{\varepsilon^2 5\rho \ln \frac{2}{\delta}}{\varepsilon^2 \rho 5} = -\ln \frac{2}{\delta} = \ln \frac{\delta}{2}$$

und daher $\mathbf{Pr}[(1-\varepsilon)|G| \leqslant Z \leqslant (1+\varepsilon)|G|] \geqslant 1-\delta$. ∎

 © Th. Worsch 2000-2015

Der obige Satz besagt, dass man eine „gute" Approximation erhält, wenn man unter anderem N umgekehrt proportional zu $\rho = |G| / |U|$ wählt. Man beachte, dass in der eingangs betrachteten Bestimmung von #F dieser Wert 2^{-n} sein *kann*. Dies bedeutete dann eine exponentielle Laufzeit des Algorithmus.

Und „bedauerlicherweise" sind die Chernoff-Schranken recht gut. Das heißt, solch große Laufzeiten sind nicht nur hinreichend, sondern bei obigem einfachen Algorithmus auch notwendig.

Um zu einem *polynomiellen* RAS zu kommen, muss man also etwas anders vorgehen. Der problematische Fall oben liegt dann vor, wenn G sehr klein im Vergleich zu U ist. Wir werden im folgenden daher auf einen Algorithmus hinarbeiten, bei dem U von vorne herein kleiner ist. Für das DNF-Problem heißt das, dass man u. U. nicht mehr *alle* Variablenbelegungen in Betracht zieht, sondern nur manche.

10.17 Es sei V ein endliches Universum, von dem m Teilmengen $H_1, \dots, H_m \subseteq V$ derart gegeben seien, dass gilt:

- Für alle i ist $|H_i|$ in Polynomialzeit berechenbar.
- Für alle i kann man aus H_i zufällig gleichverteilt ein Element auswählen.
- Für alle $v \in V$ und alle i kann man in Polynomialzeit feststellen, ob $v \in H_i$ ist oder nicht.

Die Aufgabe besteht darin, die Größe von $H = H_1 \cup \cdots \cup H_m$ zu bestimmen.

10.18 Im Fall unseres DNF-Problems werden später als H_i diejenigen Variablenbelegungen gewählt werden, die die Klausel C_i erfüllen. Da die Formel F in DNF vorliegt, ist dann $|H|$ gerade die Gesamtzahl von Variablenbelegungen, die F erfüllen.

10.19 Es sei $U = H_1 \dot\cup \cdots \dot\cup H_m$ die disjunkte Vereinigung aller H_i, also etwa $U = \{(v, i) \mid v \in H_i\}$. Offensichtlich ist $|U| \geqslant |H|$. (Da die H_i nicht paarweise disjunkt sein müssen, werden manche Elemente mehrfach gezählt.)

Es sei $\mathrm{cov}(v) = \{(v, i) \mid (v, i) \in U\}$. Offensichtlich gilt:

- Es gibt genau $|H|$ nichtleere Mengen $\mathrm{cov}(v)$.
- Die Mengen $\mathrm{cov}(v)$ sind paarweise disjunkt, partitionieren also U.
- $|U| = \sum |\mathrm{cov}(v)|$.
- Für alle $v \in V$ ist $|\mathrm{cov}(v)| \leqslant m$.

Es sei $f : U \to \{0, 1\}$ mit

$$f((v, i)) = \begin{cases} 1 & \text{falls } i = \min\{j \mid v \in H_j\} \\ 0 & \text{sonst} \end{cases}$$

und es sei $G = \{(v, i) \mid f((v, i)) = 1\}$.

Dann ist $|G| = |H|$, denn in G wird jedes $v \in H$ genau einmal gezählt, nämlich für das kleinste i mit $v \in H_i$. Zur Veranschaulichung ist die Situation in Abbildung 10.1 an einem Beispiel skizziert.

Um die Größe von $H \subseteq V$ zu bestimmen, kann man also auch die Größe von $G \subseteq U$ berechnen. Der entscheidende Punkt ist, dass im letzteren Fall die Größenverhältnisse „günstiger" sind.

10.20 Lemma. $\rho = |G| / |U| \geqslant 1/m$.

Zum Beweis muss man nur beachten, dass $|U| = \sum_{v \in H} |\mathrm{cov}(v)| \leqslant \sum_{v \in H} m = m\,|H| = m\,|G|$ ist.

10.21 Algorithmus. Wir wenden nun das obige Schema an, um das DNF-Problem zu lösen. Als H_i wähle man die Menge aller Variablenbelegungen, die Klausel C_i erfüllen, und wende den Algorithmus wie in 10.12 beschrieben auf die Mengen G und U an, wie sie in Punkt 10.19 definiert wurden.

 © Th. Worsch 2000-2015

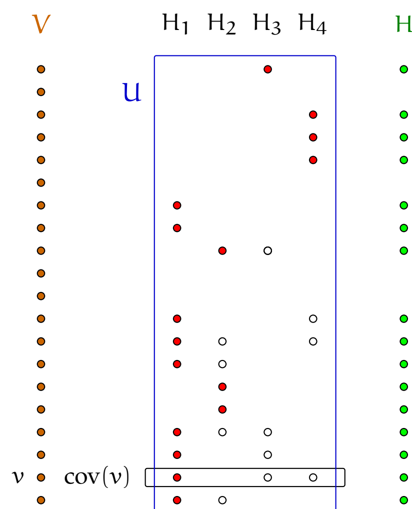


Abbildung 10.1: Ein Beispiel zu Punkt 10.19. Die Elemente von U, die zu G gehören, sind rot gekennzeichnet.

⟨$F = C_1 \vee \cdots \vee C_m$ *mit Klauseln* $C_i = l_{i,1} \wedge \cdots \wedge l_{i,r_i}$ *sei die vorgelegte Formel*⟩
⟨δ *und* ε *seien die vorgegebenen Approximationsparameter*⟩
sizeU $\leftarrow 0$
for $i \leftarrow 1$ **to** m **do**
  sizeHi $\leftarrow 2^{n-r_i}$
  sizeU $\leftarrow$ *sizeU* + *sizeHi*
od
$N \leftarrow \frac{5m}{\varepsilon^2} \ln \frac{2}{\delta}$
$z \leftarrow 0$
for $j \leftarrow 1$ **to** N **do**
  $i \leftarrow$ ⟨*zufällig aus* $[1; m]$ *mit Wahrscheinlichkeit sizeHi/sizeU ausgewähltes Element*⟩
  $x \leftarrow$ ⟨*Variablenbelegung; zufällig gleichverteilt ausgewählt aus denen, die Klausel* C_i *erfüllen*⟩
  if $\neg\exists j < i : x \in H_j$ **then** ⟨$x \in G$?⟩
    $z \leftarrow z + 1$
  fi
od
return $\frac{z}{N} \cdot$ *sizeU*

10.22 Satz. *Algorithmus 10.21 ist ein* (ε, δ)*-FPRAS für das DNF-Problem.*

10.23 Beweis. Die Größe der Eingabe liegt in $\Omega(n+m)$ und in $O(nm)$. Nach Lemma 10.20 ist $\rho \geqslant 1/m$. Wir wollen nun Satz 10.15 anwenden, um folgern zu können, dass Algorithmus 10.21 tatsächlich ein (ε, δ)-FPRAS für das DNF-Problem ist.

Als H_i wird die Menge aller Variablenbelegungen benutzt, die Klausel C_i erfüllen. Es ist also $|H_i| = 2^{n-r_i}$. Damit muss noch bewiesen werden, dass in der zweiten Schleife durch die Zuweisungen an i und x stets zufällig ein Element aus U *gleichverteilt* ausgewählt wird. Die Wahrscheinlichkeit, ein bestimmtes (x, i) auszuwählen, ist aber gerade $|H_i| / \sum |H_i| \cdot 1/|H_i| = 1/\sum |H_i| = 1/|U|$.

 © Th. Worsch 2000-2015

Also ist Algorithmus 10.21 tatsächlich ein Spezialfall von Algorithmus 10.12.

Die Anzahl Schleifendurchläufe ist $N = \frac{5m}{\varepsilon^2} \ln \frac{2}{\delta}$, also polynomial in $1/\varepsilon$, $\log 1/\delta$ und in der Problemgröße. Wegen der Voraussetzungen aus Punkt 10.19 ist daher auch die Gesamtlaufzeit polynomial in den genannten Parametern. ∎

 © Th. Worsch 2000-2015

11 Online-Algorithmen am Beispiel des Seitenwechselproblems

11.1 Unter *Online-Algorithmen* versteht man Algorithmen, denen nicht schon zu Beginn der Berechnung die ganze Eingabe vollständig zur Verfügung gestellt wird, sondern Schritt für Schritt in einer Folge $P = (r_1, r_2, \ldots, r_n)$ von „Anforderungen". Auf jede Anforderung muss der Online-Algorithmus mit einer Aktion reagieren (ohne zu wissen, wie die weiteren Anforderungen aussehen werden). Einmal gefällte Entscheidungen können nicht zurück genommen werden.

Im ersten Abschnitt werden wir kurz auf deterministische Online-Algorithmen für ein konkretes Problem eingehen, das des „Seitenwechsels" bei schnellen Zwischenspeichern. Dabei werden wir grundlegende Begriffe einführen und prinzipielle Unterschiede zur klassischen Art algorithmischer Aufgabenstellung kennen lernen. Im zweiten Abschnitt werden randomisierte Online-Algorithmen eingeführt und das Konzept der „Widersacher" oder „Adversaries". Im dritten Abschnitt stellen wir einen randomisierten Online-Algorithmus vor, der (gegen „unwissende" Widersacher) signifikant besser ist als es jeder deterministische sein kann. Seitenwechsel gegen „adaptive" Widersacher ist Gegenstand des vierten Abschnittes.

Wer genauer an diesem Thema interessiert ist, dem sei das Buch von Allan Borodin und El-Yaniv (1998) empfohlen.

11.1 Das Seitenwechselproblem und deterministische Algorithmen

11.2 Dem *Seitenwechselproblem* liegt die folgende Aufgabenstellung zu Grunde: Ein Rechner ist mit einem *Cache* der Größe k und einem Hauptspeicher größeren Umfangs N ausgestattet. Der Zugriff auf Daten im Cache ist schneller und daher zu bevorzugen. Jede Anforderung r wird durch die Adresse einer Hauptspeicherzelle repräsentiert, deren Inhalt man lesen möchte. Die Aufgabe besteht darin, dafür zu sorgen, dass möglichst oft die zu lesenden Daten schon im Cache liegen und möglichst selten ein *Cache Miss* auftritt, bei dem doch auf den Hauptspeicher zugegriffen werden muss. Soll das dabei gelesene Datum im Cache abgelegt werden, ergibt sich bei bereits gefülltem Cache das Problem, ein früher dort abgelegtes Datum auszuwählen und durch das neue zu ersetzen.

Die Qualität eines Algorithmus A für das Seitenwechselproblem ist die Anzahl $f_A(r_1, \ldots, r_n)$ der bei einer Anforderungsfolge $(r_1, \ldots, r_n)$ insgesamt auftretenden Cache Misses.

11.3 Sogenannte *Offline-Algorithmen* für das Seitenwechselproblem dürfen für die Auswahl eines Datums, das bei einem Cache Miss bei Anforderung r_i aus einem vollen Cache entfernt werden soll, die Kenntnis auch aller noch folgenden Anforderungen r_j mit $j > i$ benutzen.

Man kann zeigen, dass die Anzahl Cache Misses minimiert wird, wenn immer (sofern nötig) das Datum aus dem Cache entfernt wird, das am spätesten in der Zukunft jemals wieder benötigt wird. Dieser Algorithmus heißt üblicherweise **MIN**. Der Beweis seiner Optimalität ist nicht trivial (Belady 1966; Mattison u. a. 1971).

11.4 Definition Die Anzahl der bei einer Anforderungsfolge $(r_1, \dots, r_n)$ insgesamt auftretenden Cache Misses eines optimalen Offline-Algorithmus bezeichnen wir mit $f_O(r_1, \dots, r_n)$. ◇

11.5 Betrachten wir für einen Moment den Fall, dass es nur ein Datum mehr gibt als in den Cache passen. Man kann zeigen, dass *in diesem Fall* die Anzahl der Cache Misses des Offline-Algorithmus **MIN** für Folgen von n Anforderungen schlimmstenfalls n/k ist.

Algorithmus **MIN** ist in der Praxis völlig unbrauchbar, weil eine CPU bei der Cacheverwaltung eben *nicht* weiß, welche Anforderungen in der Zukunft kommen werden. Wir wenden uns daher nun Online-Algorithmen zu, die im Falle eines Cache Miss bei Anforderung r_i nur auf Grund der Kenntnis von $r_1, \dots, r_i$ entscheiden, welches Datum aus dem Cache verdrängt wird.

11.6 Das Verhalten eines Online-Algorithmus bei der Verarbeitung einer Anforderung r_i ist unabhängig von r_{i+1}. Ist der Cache gefüllt, kann man folglich (in dem einzig interessanten Fall $N > k$) erzwingen, dass *jede* Anforderung zu einem Cache Miss führt.

Daher gibt es im Fall $N = k + 1$ auch beliebig lange Anforderungsfolgen, für die jeder Online-Algorithmus k mal mehr Cache Misses produziert als der optimale Offline-Algorithmus **MIN**.

11.7 Üblicherweise beschreibt man die Qualität eines Algorithmus, indem man für jedes n den Aufwand für die „schlimmsten" Eingaben dieser Länge angibt (*worst case complexity*). Aus der vorangegangenen Bemerkung ergibt sich, dass beim Seitenwechselproblem diese vergröbernde Sichtweise sinnlos ist. Denn man kann beliebig lange Anforderungsfolgen konstruieren, bei denen *in jedem Schritt* ein Cache Miss auftritt.

Ein sinnvoller Vergleich von Algorithmen ist so also nicht möglich. Der naheliegende Ausweg ist, jede Probleminstanz (i. e. Anforderungsfolge) einzeln zu betrachten.

Für den Vergleich von deterministischen (Online-)Algorithmen hat es sich als sinnvoll erwiesen, die folgenden Definition zu Grunde zu legen.

11.8 Definition Ein deterministischer Online-Algorithmus A heißt C-*kompetitiv*, falls es ein b gibt, das von C abhängen darf, aber nicht von n, so dass für alle Anforderungsfolgen $(r_1, \dots, r_n)$ gilt:

$$f_A(r_1, \dots, r_n) - C \cdot f_O(r_1, \dots, r_n) \leqslant b \ .$$

Der *Wettbewerbsfaktor* C_A von A ist das Infimum der C, so dass A C-kompetitiv ist. ◇

Man beachte, dass dies insofern eine strenge Forderung ist, als die Ungleichung für *alle* Anforderungsfolgen gelten muss.

11.9 Beispiel. Ein Online-Algorithmus, der zum Beispiel bei vielen Prozessoren mit mehrfach assoziativen First Level Caches (siehe z. B. Ungerer 1995) benutzt wird, ist **LRU**. Diese Abkürzung steht für *least recently used*. D. h., wenn ein Datum aus dem Cache verdrängt werden muss, wird jeweils das ausgewählt, für das am längsten keine Anforderung mehr auftrat.

Eine andere naheliegende Vorgehensweise ist **FIFO**, d. h. man verdrängt das Datum, das von den derzeit im Cache vorhandenen am frühesten angefordert wurde.

11.10 Daniel D. Sleator und Robert E. Tarjan (1985) haben gezeigt, dass **LRU** und **FIFO** k-kompetitiv sind. Andererseits folgt aus dem in Punkt 11.5 und Punkt 11.6 Gesagten, dass kein deterministischer Online-Algorithmus besser als k-kompetitiv sein kann. Die genannten Algorithmen sind also optimal.

 © Th. Worsch 2000-2015

11.2 Randomisierte Online-Algorithmen und Widersacher

11.11 Bei einem randomisierten Online-Algorithmus R geht in die Wahl des aus dem Cache zu verdrängenden Datums eine zufällige Komponente mit ein. Folglich ist die Zahl der Cache Misses nun eine *Zufallsvariable* $f_R(r_1, \dots, r_n)$.

Im deterministischen Fall gibt der Kompetitivitätsfaktor eines Algorithmus A an, um wieviel mal A schlechter als der optimale Algorithmus schlimmstenfalls ist. Etwas Ähnliches möchte man auch im randomisierten Fall.

Eine einfache Möglichkeit bestände darin, z. B. den Erwartungswert von f_R wie in Definition 11.8 mit f_O von **MIN** zu vergleichen. In Satz 11.29 werden wir (ohne Beweis) sehen, warum man das *nicht* einfach so machen möchte.

11.12 Die Vorstellung, die sich für den Nachweis der Existenz solch ungünstiger Fälle eingebürgert hat, ist die, dass ein „böser Widersacher" (engl. *adversary*) eine schlechte Anforderungsfolge erzeugt, dessen Länge er als Eingabe bekommt. Für den Widersacher selbst ist auch ein Verfahren festgelegt, nach dem die für ihn entstehenden Kosten $f_W(r_1, \dots, r_n)$ bestimmt werden. Sie übernehmen die Rolle der Kosten von **MIN** im deterministischen Fall.

Es gibt verschiedene Varianten von Widersachern. Alle kennen den Programmtext des randomisierten Algorithmus R, gegen den sie arbeiten sollen. Es stellt sich nun aber auch die Frage, wieviel Information ihnen über die Werte von Zufallsbits zur Verfügung steht.

- Ein *unwissender Widersacher* W (der englische Begriff *oblivious adversary* heißt eigentlich vergesslicher Widersacher) ist einer, der *kein* Wissen über die Zufallsbits hat. Zu vorlegtem randomisierten Algorithmus R und n wird W also immer die gleiche Folge $(r_1, \dots, r_n)$ erzeugen.
- Im Gegensatz dazu arbeitet ein sogenannter adaptiver Widersacher gegen eine konkrete Abarbeitung eines randomisierten Algorithmus R. Hat er bereits $(r_1, \dots, r_i)$ erzeugt, so kann er für die Bestimmung von r_{i+1} auch auf die Information zurückgreifen, welche Zufallsbits R gewürfelt hat und welche Daten sich in Folge dessen momentan im Cache von R befinden.

Man stellt sich nun auch die Frage, womit man die Anzahl der Cache Misses des zu beurteilenden Algorithmus R vergleicht. Es hat sich eingebürgert, die folgenden Varianten zu betrachten:

- Bei einem unwissenden Widersacher werden die Kosten für die Bearbeitung der Anforderungsfolge mit dem optimalen deterministischen Algorithmus mit nur $f_O(r_1, \dots, r_n)$ Cache Misses in Rechnung gestellt.
- Bei adaptiven Widersachern unterscheidet nun zwei Varianten.
  - Ein *adaptiver Online-Widersacher* (engl. *adaptive online adversary*) muss nach der Erzeugung jedes r_i sofort, also unter Benutzung eines Online-Algorithmus, entscheiden, welches andere Datum dadurch aus dem Cache verdrängt werden soll (falls das nötig ist).
  - Ein *adaptiver Offline-Widersacher* (engl. *adaptive offline adversary*) kann zunächst die ganze Anforderungsfolge $(r_1, \dots, r_n)$ erzeugen. Für f_W werden die Kosten für die Bearbeitung dieser Anforderungsfolge durch den optimalen Offline-Algorithmus in Rechnung gestellt.

 © Th. Worsch 2000-2015

In beiden Fällen ist nun auch $f_O(r_1, \dots, r_n)$ eine Zufallsvariable, denn die erzeugte Anfragefolge hängt von der Wahl der Zufallsbits von R ab. (Man mache sich klar, dass dies auch für adaptive Offline-Widersacher gilt.)

11.13 Definition

- Ein randomisierter Online-Algorithmus R ist C-*kompetitiv gegen unwissende Widersacher*, wenn es ein von n unabhängiges b gibt, so dass für jede Anforderungsfolge $(r_1, \dots, r_n)$ gilt:

$$\mathbf{E}\left[f_R(r_1, \dots, r_n)\right] - C \cdot f_O(r_1, \dots, r_n) \leqslant b$$

  Das Infimum solcher C wird auch mit C_R^{obl} bezeichnet.
- Ein randomisierter Online-Algorithmus R ist C-*kompetitiv gegen einen adaptiven Online-Widersacher*, wenn es ein von n unabhängiges b gibt, so dass gilt:

$$\mathbf{E}\left[f_R(r_1, \dots, r_n) - C \cdot f_W(r_1, \dots, r_n)\right] \leqslant b$$

  Das Infimum solcher C wird auch mit C_R^{aon} bezeichnet.
- Ein randomisierter Online-Algorithmus R ist C-*kompetitiv gegen einen adaptiven Offline-Widersacher*, wenn es ein von n unabhängiges b gibt, so dass gilt:

$$\mathbf{E}\left[f_R(r_1, \dots, r_n) - C \cdot f_O(r_1, \dots, r_n)\right] \leqslant b$$

  Das Infimum solcher C wird auch mit C_R^{aof} bezeichnet. ◇

Bei den Online-Widersachern ergeben sich aus den Zufallsentscheidungen des randomisierten Algorithmus jeweils zufällige Anforderungsfolgen $(r_1, \dots, r_n)$. Sie sind in den beiden zuletzt genannten Ungleichungen jeweils zweimal erwähnt, um deutlich zu machen, dass davon die Werte der Zufallsvariablen abhängen.

11.3 Seitenwechsel gegen einen unwissenden Widersacher

Wir beschreiben nun einen Algorithmus R, von dem gezeigt werden wird, dass er $2H_k$-kompetitiv gegen unwissende Widersacher ist. Anschließend werden wir nachweisen, dass er damit bis auf einen konstanten Faktor 2 an eine untere Schranke herankommt.

11.14 Algorithmus. (Marker-Algorithmus von Fiat u. a. (1991))

```
⟨Cache: Speicherstellen cache[i] mit 1 ⩽ i ⩽ k,⟩
⟨       die jeweils mit einem Markierungsbit mark[i] versehen sind.⟩

⟨Initialisierung:⟩
for i ← 1 to k do mark[i] ← 0 od

⟨Abarbeitung der Anforderungen:⟩
while ⟨noch weitere Anforderungen⟩ do
    r ← ⟨nächste Anforderung⟩
    if ⟨memory[r] ist nicht im Cache⟩ then
        if ⟨alle mark[i] = 1⟩ then ⟨alle mark[i] ← 0⟩ fi
```

 © Th. Worsch 2000-2015

$i \leftarrow \langle$*zufällig gleichverteilt gewähltes* j *derer mit mark*$[j] = 0\rangle$
cache$[i] \leftarrow$ *memory*$[r]$
else
$i \leftarrow \langle$*Index mit cache*$[i] =$ *memory*$[r]\rangle$
fi
mark$[i] \leftarrow 1$
od

11.15 Satz. *Algorithmus 11.14 ist* $2H_k$*-kompetitiv gegen unwissende Widersacher.*

11.16 Beweis. Der Ablauf des obigen Algorithmus zerfällt in sogenannte *Phasen*. Jede Phase beginnt mit dem Zurücksetzen aller Markierungsbits auf 0 und endet unmittelbar vor Beginn der nächsten Phase.

Wir untersuchen nun die Arbeit des optimalen Offline-Algorithmus und des Markeralgorithmus für eine Anforderungsfolge $r_1, r_2, \dots, r_n$. Zu Beginn seien die Cacheinhalte für beide Algorithmen gleich und es führe r_1 zu einem Cache Miss. Folglich beginnt jede Phase mit einem Cache Miss. Geschieht dieser für Anforderung r_i, so ist die Anforderungsteilfolge für die gesamte Phase die maximale Teilfolge $r_i, \dots, r_j$, während der genau k mal ein Markierungsbit auf 1 gesetzt wird.

Wir betrachten nun eine einzelne beliebige Phase. Ein Datum heiße *markiert*, wenn es *zum betrachteten Zeitpunkt* an einer markierten Stelle des Caches liegt. Der Markeralgorithmus entfernt bei einem Cache Miss stets ein nicht markiertes Datum aus dem Cache, und das den Cache Miss verursachende Datum ist ab dem Zeitpunkt, zu dem es in den Cache geladen wird, markiert.

Ein Datum heiße *veraltet*, wenn es *zu Beginn der Phase* im Cache ist und es heiße *sauber*, wenn es *zu Beginn der Phase nicht* im Cache ist. Die Anforderung eines sauberen Datums führt also auf jeden Fall zu einem Cache Miss. Die Anforderung eines veralteten Datums kann ebenfalls zu einem Cache Miss führen, nämlich dann, wenn es zwischenzeitlich (aufgrund eines anderen Cache Miss) aus dem Cache entfernt worden war. Die Anforderung eines veralteten Datums kann aber *nur einmal* innerhalb einer Phase zu einem Cache Miss führen, da es beim erneuten Laden markiert wird. Außerdem wird hierdurch wieder ein (nicht markiertes, also) veraltetes Datum verdrängt.

Es sei ℓ die Anzahl sauberer Datenelemente, die durch den Marker-Algorithmus im Laufe der Phase (unter Umständen mehrfach) angefordert werden.

Wir zeigen nun zweierlei:

1. Der optimale Offline-Algorithmus führt im Mittel pro Phase zu mindestens $\ell/2$ Cache Misses.

2. Beim Marker-Algorithmus ist die erwartete Anzahl von Cache Misses pro Phase ℓH_k.

Hieraus folgt die Behauptung des Satzes.

zu 1. Es bezeichne S_O die Menge der vom optimalen Algorithmus und S_M die der vom Marker-Algorithmus im Cache gehaltenen Elemente. Es sei d_a die Größe von $S_O \setminus S_M$ zu Beginn und d_e die Größe am Ende der Phase. Es sei m_O die Anzahl der Cache Misses des optimalen Algorithmus während der Phase.

Zu Beginn der Phase sind die ℓ später durch den Marker-Algorithmus angeforderten sauberen Datenelementen nicht in S_M. Und höchstens d_a von ihnen sind in S_O. Also ist $m_O \geqslant \ell - d_a$.

© Th. Worsch 2000-2015

Am Ende der Phase besteht S_M genau aus den k markierten, also insbesondere auch während der Phase angeforderten Elementen. Da davon d_e am Ende nicht mehr in S_O sind, muss der optimale Algorithmus sie wieder verdrängt haben. Das kann nur durch den Cache Miss eines anderen Elementes geschehen sein. Als muss der optimale Algorithmus mindestens d_e Cache Misses erzeugt haben: $m_O \geqslant d_e$.

Insgesamt ist also $m_O \geqslant \max\{\ell - d_a, d_e\} \geqslant (\ell - d_a + d_e)/2$.

Der Wert d_e am Ende einer Phase ist der Wert d_a zu Beginn der nächsten. Summiert man über alle Phasen, heben sich von den Brüchen die Summanden für d_a und d_e auf. Ausnahme sind der Wert d_a zu Beginn der ersten Phase, der nach Voraussetzung $d_a = 0$ ist, und der Wert d_e am Ende der letzten Phase. Hieraus folgt, dass im Mittel pro Phase $m_O \geqslant \ell/2$ ist.

zu 2. Die jeweils erste Anforderung eines der ℓ sauberen Elemente führt zu einem Cache Miss; die übrigen Anforderungen sauberer Elemente führen nicht zu einem Cache Miss. Alle anderen Anforderungen betreffen veraltete Elemente, von denen am Ende der Phase $k - \ell$ im Cache sind. Die erwartete Anzahl der hierdurch erzwungenen Cache Misses wird maximiert, wenn zuerst alle sauberen Elemente angefordert werden. Danach fehlen im Cache ℓ veraltete Elemente. Dies bleibt bis zum Ende der Phase so, denn durch Anforderung eines nicht mehr vorhandenen veralteten Datums x wird stets ein anderes veraltetes Datum x' verdrängt. Aber x wird bis zum Ende der Phase nicht mehr verdrängt.

Es seien nun $x_1, \dots, x_{k-\ell}$ die am Ende der Phase im Cache befindlichen veralteten Elemente. Dabei sei x_1 dasjenige, das von allen x_i als erstes während der Phase angefordert wurde, x_2 dasjenige, das als zweites angefordert wurde, und so weiter. Die Wahrscheinlichkeit, dass bei der ersten Anforderung von x_i $(1 \leqslant i \leqslant k - \ell)$ ein Cache Miss auftritt, ist gleich der Wahrscheinlichkeit, dass ein nicht im Cache vorhandenes veraltetes Element aus den veralteten Elementen, die in dieser Phase noch nicht angefordert wurden, ausgewählt wird. Es sind stets ℓ veraltete Elemente nicht im Cache. Und bei der ersten Anforderung von x_i wurden $k - (i - 1)$ veraltete Elemente noch nicht angefordert. Die relative Häufigkeit eines Cache Miss ist also $\ell/(k - i + 1)$.

Es ist $\sum_{i=1}^{k-\ell} \ell/(k - i + 1) = \ell\left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{l+1}\right) = \ell(H_k - H_\ell)$. Folglich ist die erwartete Anzahl Cache Misses kleiner oder gleich $\ell + \ell(H_k - H_\ell) = \ell H_k - (H_\ell - 1)\ell \leqslant \ell H_k$.

■

Der durch Algorithmus 11.14 gegebenen oberen Schranke für C_R^{obl} stellen wir nun eine untere Schranke gegenüber, die zuerst von Fiat u. a. (1991) gezeigt wurde.

11.17 SATZ. *Es sei R ein randomisierter Algorithmus für das Seitenwechselproblem. Dann ist* $C_R^{obl} \geqslant H_k = \sum_{i=1}^{k} 1/i \in \Theta(\log k)$.

11.18 BEWEIS. Wir gehen in zwei Schritten vor:

1. Es sei $\mathbf{p}$ eine Wahrscheinlichkeitsverteilung für Folgen von Anforderungen. Für einen deterministischen Online-Algorithmus A für das Seitenwechselproblem sei seine Kompetitivität $C_A^{\mathbf{p}}$ unter $\mathbf{p}$ das Infimum aller C, so dass eine von n unabhängige Konstante b existiert mit $\mathbf{E}[f_A(r_1, \dots, r_n)] - C \cdot \mathbf{E}[f_O(r_1, \dots, r_n)] \leqslant b$ für alle Anforderungsfolgen $(r_1, \dots, r_n)$.

   Mit Hilfe von Überlegungen ganz ähnlich denen, die der Minimax-Methode von Yao zu Grunde liegen, kann man zeigen:

$$\inf_R C_R^{obl} = \sup_{\mathbf{p}} \inf_A C_A^{\mathbf{p}} .$$

 © Th. Worsch 2000-2015

Mit anderen Worten ist C^{obl} gleich der Kompetitivität eines besten deterministischen Online-Algorithmus für eine „worst case"-Anforderungsfolge.

Diese hier nicht bewiesene Tatsache werden wir nun benutzen,

2. und zwar so: Es wird eine Wahrscheinlichkeitsverteilung $\mathbf{p}$ für Anforderungsfolgen konstruiert, so dass für die Erwartungswerte gilt, dass der Algorithmus **MIN** (Offline!) H_k mal weniger Cache Misses hat als jeder deterministische Online-Algorithmus. Der Ausdruck $\inf_A C_A^{\mathbf{p}}$ auf der rechten Seite der Gleichung ist für dieses $\mathbf{p}$ also mindestens H_k.

   Die Größe des Cache werde wie immer mit k bezeichnet, und es werden $k+1$ verschiedene Anforderungen $I = \{a_1, \dots, a_{k+1}\}$ benutzt. Die Wahrscheinlichkeitsverteilung $\mathbf{p}$ für die Anforderungsfolgen $(r_1, \dots)$ ergibt sich durch die folgende „zufällige Konstruktion" einer solchen Folge: Liegen $r_1, \dots, r_{i-1}$ schon fest, so ergibt sich r_i, indem es zufällig gleichverteilt aus $I \setminus \{r_{i-1}\}$ ausgewählt wird. Außerdem werde r_1 zufällig gleichverteilt aus I ausgewählt.

   Jede Anforderungsfolge kann wie folgt in *Runden* aufgeteilt werden[1]: Die erste Runde beginnt mit r_1 und jede weitere Runde beginnt unmittelbar nach Ende der vorangegangenen Runde. Jede Runde endet unmittelbar *bevor zum ersten Mal jedes* der $k+1$ existierenden $a_j \in I$ mindestens einmal angefordert worden ist. Innerhalb einer Runde finden sich also stets Anforderungen von k verschiedenen a_j und unter diesen befindet sich *nicht* das erste angeforderte Element der nächsten Runde.

   O. B. d. A. führe r_1 zu einem Cache Miss. Algorithmus **MIN** entfernt aus seinem Cache immer dasjenige Element, das am spätesten in der Zukunft wieder angefordert wird. Dies ist also das Element, das als erstes in der zweiten Runde angefordert wird. Per Induktion ergibt sich, dass bei **MIN** in jeder Runde nur genau ein Cache Miss auftritt, nämlich bei der ersten Anforderung.

   Wie lange dauert eine Runde? Stellt man sich die $a_j \in I$ als Knoten des vollständigen Graphen K_{k+1} vor, dann entspricht wegen der oben beschriebenen Wahl der r_i jede Anforderungsfolge einem Random Walk in K_{k+1}. Der Erwartungswert für die Länge einer Runde ist gerade der Erwartungswert für die Anzahl Schritte, bis ein Random Walk, der an einem Knoten beginnt, jeden Knoten mindestens einmal besucht hat (also die sogenannte Überdeckungszeit). Dieser Erwartungswert ist für vollständige Graphen gerade kH_k.

   Wieviele Cache Misses erzeugt ein deterministischer Online-Algorithmus A? Zu jedem Zeitpunkt gibt es genau ein a_j, das nicht im Cache von A ist. Dieses a_j ist mit Wahrscheinlichkeit $1/k$ die nächste Anforderung. Also ist der Erwartungswert für die Anzahl Cache Misses während einer Runde (deren Länge Erwartungswert kH_k hat) dann gerade H_k, während in der gleichen Zeit **MIN** nur 1 Cache Miss erzeugt.

■

Ein Algorithmus, der noch um einen Faktor 2 besser ist als Algorithmus 11.14 wurde von McGeoch und Daniel Dominic Sleator (1991) angegeben. Er ist wegen des eben gezeigten Satzes 11.17 gegen unwissende Widersacher optimal.

[1] Dies ist mal wieder eine Stelle, an der wir bewusst vom von Motwani und Raghavan (1995, S. 375) gegebenen Beweis abweichen.

 © Th. Worsch 2000-2015

11.4 Einschub: Amortisierte Analyse

Solange dieser Abschnitt noch nicht (fertig) geschrieben ist, lese man folgende Dokumente:

- Robert Endre Tarjan (1985). „Amortized Computational Complexity“. In: *SIAM Journal Alg. Disc. Meth.* 6.2, S. 306–318
- Das „Handout“ `http://www.ugrad.cs.ubc.ca/~cs320/2010W2/handouts/aa-nutshell.pdf` zu der Vorlesung „CPSC 320: Intermediate Algorithm Design and Analysis“ von Patrice Belleville
- `http://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf`
- Die “Lecture Notes” zu Lecture 16 (Amortized Analysis) der Vorlesung „Algorithms“ (CS357) gehalten von Vijaya Ramachandran im Frühjahr 2006, zu finden noch über `www.archive.org` durch Suche nach `http://www.cs.utexas.edu/~vlr/s06.357/notes/lec16.pdf` (und unter `http://www.coursehero.com/file/4742/Lecture-16/`).

Amortisierte Analyse ist eine Methode, um für eine ganze Folge von Operationen eine obere Schranke für die „Kosten“ für ihre Ausführung erhalten. Das Ziel ist dabei eine bessere Schranke zu finden als die Anzahl der Operationen mal schlimmste Kosten einer einzelnen Operation. Als Kosten wird häufig die Laufzeit herangezogen; im vorliegenden Kapitel geht es natürlich um die Anzahl Cache Misses.

Gegeben sei ein Stack, auf die üblichen Operationen PUSH und POP sowie für $j \in \mathbb{N}_0$ „Multi-Pop-Operationen“ MPOP(j) ausgeführt werden können. PUSH und POP mögen jeweils einen Schritt benötigen. MPOP(j) macht j POP -Operationen (sofern der Stack groß genug ist) und benötigt daher im allgemeinen j Schritte.

Eine zugegebenermaßen *sehr* naive Worst-Case-Analyse für den Zeitbedarf einer Folge von n Operationen würde $n \cdot n$ liefern, da es MPOP -Operationen geben kann, die $\Omega(n)$ Zeit brauchen.

Der Zustand der Datenstruktur nach i Operationen wird mit D_i bezeichnet. Wir definieren nun eine sogenannte *Potenzialfunktion* $\Phi(i)$ ($= \Phi(D_i)$) für die Datenstruktur, die die folgenden Eigenschaften hat:

- $\Phi(0) = 0$
- für alle i ist $\Phi(i) \geqslant 0$

Im Falle des Stacks wähle man als $\Phi(i)$ die Größe des Stacks.

Es bezeichne nun r_i die *realen* Kosten der i-ten Operation und a_i ihre sogenannten *amortisierten* Kosten. Dabei ist per definitionem

$$a_i = r_i + \Phi(i) - \Phi(i-1)$$

Man kann sich $\Phi(i)$ als ein Guthaben auf einem Konto vorstellen (das man nicht überziehen darf). Ist $\Phi(i) > \Phi(i-1)$ dann zahlt man zusätzlich zu den realen Kosten etwas auf das Konto ein, und im Fall $\Phi(i) < \Phi(i-1)$ wird ein Teil der realen Kosten vom Guthaben abgehoben.

 © Th. Worsch 2000-2015

Die amortisierten Gesamtkosten sind dann

$$\begin{aligned}
\sum_{i=1}^{n} a_i &= \sum_{i=1}^{n} (r_i + \Phi(i) - \Phi(i-1)) \\
&= \sum_{i=1}^{n} r_i + \sum_{i=1}^{n} \Phi(i) - \sum_{i=1}^{n} \Phi(i-1) \\
&= \sum_{i=1}^{n} r_i + \sum_{i=1}^{n} \Phi(i) - \sum_{i=0}^{n-1} \Phi(i) \\
&= \sum_{i=1}^{n} r_i + \Phi(n) - \Phi(0)
\end{aligned}$$

Hieraus ergibt sich mit $\Phi(0) = 0$:

$$\sum_{i=1}^{n} r_i = \sum_{i=1}^{n} a_i - \Phi(n) \leqslant \sum_{i=1}^{n} a_i$$

Die realen Gesamtkosten sind also *höchstens* so hoch wie die amortisierten Kosten. Wie man gleich sehen wird, gibt es durchaus Fälle, in denen letztere einfacher und besser abzuschätzen sind, so dass man eine bessere obere Schranke für die realen Kosten erhält. Das Kunststück besteht darin, eine für den jeweiligen Anwendungsfall passende Potenzialfunktion zu finden.

Wie verändert sich bei den einzelnen Operationen das Potenzial im Beispiel mit dem Stack und wie groß sind die amortisierten Kosten?

- Push: Der Stack wächst von ℓ auf $\ell+1$, also ist $a_i = 1 + (\ell+1) - \ell = 2$.

  Der „Trick" besteht hier also darin, für jedes Push zusätzlich zu den realen Kosten eine Einheit auf das Guthabenkonto „einzuzahlen". Wofür das gut ist, sieht man gleich:

- Pop: Der Stack schrumpft von ℓ auf $\ell-1$, also ist $a_i = 1 + (\ell-1) - \ell = 0$.

  Das Sparen hat sich gelohnt: Es ist bestimmt genug auf dem Konto, um das Pop zu bezahlen. Das Gleiche gilt im letzten Fall:

- Mpop(j): Der Stack schrumpft von ℓ auf $\max(\ell-j, 0)$, also ist

$$\begin{aligned}
a_i &= r_i + \max(\ell-j, 0) - \ell \\
&= \begin{cases} j + \ell - j - \ell & \text{falls } \ell \geqslant j \\ \ell - \ell & \text{falls } \ell < j \end{cases} \\
&= 0
\end{aligned}$$

Damit sind also die amortisierten Kosten *aller* Operationen durch eine Konstante beschränkt. Folglich ist $\sum_{i=1}^{n} a_i \in O(n)$. Da außerdem $\Phi(n) \leqslant n$ gilt ist auch $\sum_{i=1}^{n} r_i \in O(n)$.

11.5 Seitenwechsel gegen adaptive Widersacher

11.19 Im folgenden wird die folgende Verallgemeinerung des Seitenwechselproblems betrachtet: Jedem Element x ist ein *Gewicht* $w(x) > 0$ zugeordnet. Wird ein Element x in den Cache geladen,

 © Th. Worsch 2000-2015

verursacht das Kosten in Höhe von $w(x)$. Der Gesamtaufwand für eine Anforderungsfolge ist die Summe der Kosten für die einzelnen Elemente.

Wählt man alle Gewichte identisch (z. B. zu 1), dann ergibt sich offensichtlich das weiter vorne betrachtete einfache Seitenwechselproblem.

Bei dem neuen Problem sind unter Umständen neue Algorithmen angebracht, denn es erscheint plausibel, bevorzugt Elemente mit kleinem Gewicht aus dem Cache zu verdrängen, da deren erneutes Laden „billiger“ ist.

11.20 ALGORITHMUS. (REZIPROK-ALGORITHMUS) Sind $x_1, \dots, x_k$ die zu einem Zeitpunkt im Cache befindlichen Elemente und muss eines verdrängt werden, dann wählt dieser Algorithmus dafür mit Wahrscheinlichkeit

$$\frac{1/w(x_i)}{\sum_{j=1}^{k} 1/w(x_j)}$$

Element x_i aus. „Leichte“ Elemente werden also bevorzugt.

Wir wollen nun zeigen:

11.21 SATZ. *Der Reziprok-Algorithmus ist k-kompetitiv gegen adaptive Online-Widersacher.*

11.22 BEWEIS. (VON SATZ 11.21) Es bezeichne R den Reziprok-Algorithmus und W einen adaptiven Online-Widersacher. S_i^R sei die Menge der Elemente, die sich nach der Bearbeitung der i-ten Anforderung im Cache von R befinden, und S_i^W die Menge der Elemente, die sich nach der Bearbeitung der i-ten Anforderung im Cache von W befinden.

Es bezeichne f_i^R resp. f_i^W die durch den jeweiligen Algorithmus bei der Abarbeitung der i-ten Anforderung verursachten Kosten. Die Aufgabe besteht also darin, zu zeigen, dass

$$\sum_i \left(\mathbf{E}\left[f_i^R\right] - k\mathbf{E}\left[f_i^W\right] \right)$$

beschränkt ist. Dazu definieren wir zunächst eine sogenannte Potenzialfunktion

$$\Phi_i = \sum_{z \in S_i^R} w(z) - k \sum_{z \in S_i^R \setminus S_i^W} w(z)$$

und betrachten die Zufallsvariablen $X_i = f_i^R - k f_i^W - (\Phi_i - \Phi_{i-1})$ für $1 \leqslant i \leqslant n$. Für sie gilt:

$$\sum_i X_i = \Phi_0 - \Phi_n + \left(\sum_i f_i^R - k f_i^W \right) .$$

Daher genügt es, zu zeigen, dass $\mathbf{E}\left[\sum_i X_i\right] \leqslant 0$ ist.

Wir zeigen, dass sogar für jedes einzelne i gilt: $\mathbf{E}[X_i] \leqslant 0$. Dazu stellen wir uns vor, dass jede Anforderung zuerst von W und dann von R bearbeitet wird, und untersuchen die Veränderungen, die X_i bei der Bearbeitung der Anforderung durch W bzw. R nacheinander erfährt. Um die Veränderung von Φ nach dem ersten Teilschritt zu erfassen, definieren wir noch

$$\Psi_i = \sum_{z \in S_{i-1}^R} w(z) - k \sum_{z \in S_{i-1}^R \setminus S_i^W} w(z)$$

Man beachte die Indizes in den Definitionen von Φ_i und Ψ_i.

 © Th. Worsch 2000-2015

Es ist:

$$\begin{aligned} X_i &= f_i^R - kf_i^W - (\Phi_i - \Phi_{i-1}) \\ &= f_i^R - kf_i^W - ((\Phi_i - \Psi_i) + (\Psi_i - \Phi_{i-1})) \\ &= \left(f_i^R - (\Phi_i - \Psi_i)\right) + \left(-kf_i^W - (\Psi_i - \Phi_{i-1})\right) \end{aligned}$$

Um zu zeigen, dass $\mathbf{E}[X_i] \leqslant 0$ ist, beweisen wir nacheinander:

1. $\mathbf{E}\left[-kf_i^W - (\Psi_i - \Phi_{i-1})\right] \leqslant 0$
2. $\mathbf{E}\left[f_i^R - (\Phi_i - \Psi_i)\right] \leqslant 0$

1. Widersacher W**:** Bei einem Cache Hit ist $-kf_i^W + (\Psi_i - \Phi_{i-1}) = 0$.

Betrachten wir nun den Fall eines Cache Miss von W. Die Gesamtkosten ändern sich nur um einen konstanten Betrag, wenn man bei der Ersetzung eines Elementes x' durch durch ein neues Element x im Cache jeweils nicht die Kosten $w(x)$ sondern die Kosten $w(x')$ für W in Rechnung stellt. Wir nehmen daher an, dass in diesem Fall $f_i^W = w(x')$ und folglich $-kf_i^W = -kw(x')$ ist.

Welchen Wert hat $-(\Psi_i - \Phi_{i-1})$? Es ist $S_i^W = S_{i-1}^W \setminus \{x'\} \cup \{x\}$. Wir definieren noch die Menge $A = S_{i-1}^R \setminus (S_{i-1}^W \cup \{x\})$. Dann ist

$$\begin{aligned} -\Psi_i + \Phi_{i-1} &= -\sum_{z\in S_{i-1}^R} w(z) + k\sum_{z\in S_{i-1}^R\setminus S_i^W} w(z) + \sum_{z\in S_{i-1}^R} w(z) - k\sum_{z\in S_{i-1}^R\setminus S_{i-1}^W} w(z) \\ &= k\sum_{z\in S_{i-1}^R\setminus S_i^W} w(z) - k\sum_{z\in S_{i-1}^R\setminus S_{i-1}^W} w(z) \\ &= k\sum_{z\in A} w(z) + k\begin{cases} w(x') & \text{falls } x'\in S_{i-1}^R \\ 0 & \text{sonst} \end{cases} - k\sum_{z\in A} w(z) - k\begin{cases} w(x) & \text{falls } x\in S_{i-1}^R \\ 0 & \text{sonst} \end{cases} \\ &= k\begin{cases} w(x') & \text{falls } x'\in S_{i-1}^R \\ 0 & \text{sonst} \end{cases} - k\begin{cases} w(x) & \text{falls } x\in S_{i-1}^R \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

Auch bei einem Cache Miss von W gilt folglich: $-kf_i^W - (\Psi_i - \Phi_{i-1}) \leqslant 0$.

2. Reziprokalgorithmus R**:** Bei einem Cache Miss ist $f_i^R = w(x)$. Das durch x aus dem Cache von R verdrängte Datum werde mit x'' bezeichnet. Dann gilt (man beachte, dass $x \in S_i^W$ ist):

$$\begin{aligned} -\Phi_i + \Psi_i &= -\sum_{z\in S_i^R} w(z) + k\sum_{z\in S_i^R\setminus S_i^W} w(z) + \sum_{z\in S_{i-1}^R} w(z) - k\sum_{z\in S_{i-1}^R\setminus S_i^W} w(z) \\ &= \sum_{z\in S_{i-1}^R} w(z) - \sum_{z\in S_i^R} w(z) + k\sum_{z\in S_i^R\setminus S_i^W} w(z) - k\sum_{z\in S_{i-1}^R\setminus S_i^W} w(z) \\ &= w(x'') - w(x) - k\begin{cases} w(x'') & \text{falls } x''\in S_{i-1}^R\setminus S_i^W \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

Für die Bestimmung von $\mathbf{E}[-\Phi_i + \Psi_i]$ ist zu beachten, dass R ein Element x'' mit Wahrscheinlichkeit $(1/w(x''))/\sum_y 1/w(y)$ aus dem Cache verdrängt. Also ist

$$\begin{aligned} \mathbf{E}[-\Phi_i + \Psi_i] &= -w(x) + \sum_{x''\in S_{i-1}^R} w(x'')\frac{1/w(x'')}{\sum_y 1/w(y)} - k\sum_{x''\in S_{i-1}^R\setminus S_i^W} w(x'')\cdot\frac{1/w(x'')}{\sum_y 1/w(y)} \\ &= -w(x) + k\frac{1}{\sum_y 1/w(y)} - k\frac{\left|S_{i-1}^R\setminus S_i^W\right|}{\sum_y 1/w(y)} \end{aligned}$$

© Th. Worsch 2000-2015

Unmittelbar vor dem Cache Miss von R ist $x \in S_i^W \setminus S_{i-1}^R$, also ist $\left|S_i^W \setminus S_{i-1}^R\right| \geqslant 1$. Da beide Caches gleich groß sind, ist folglich auch $\left|S_{i-1}^R \setminus S_i^W\right| \geqslant 1$ und damit

$$\mathbf{E}\left[f_i^R - \Phi_i + \Psi_i\right] = w(x) - w(x) + k\frac{1 - \left|S_{i-1}^R \setminus S_i^W\right|}{\sum_y 1/w(y)} \leqslant 0 \,.$$

∎

Der Wettbewerbsfaktor k im vorangegangenen Satz ist optimal. Im folgenden wird noch skizziert, auf welchem Wege man das zum Beispiel einsehen kann. Dazu betrachtet man ein weiteres Problem:

11.23 Problem. (k-Server-Problem) Auf k Punkten eines metrischen Raumes (M, d) befinden sich zu jedem Zeitpunkt je ein *Server*. Eine Anforderung besteht in der Angabe eines Punktes $x \in M$. Steht gerade ein Server auf x, ist nichts zu tun. Andernfalls muss ein Server von seinem momentanen Standpunkt y nach x bewegt werden. Die hierfür anfallenden Kosten sind gerade $d(x, y)$. Die Aufgabe eines Algorithmus besteht darin, den jeweils zu bewegenden Server so auszuwählen, dass die Gesamtkosten möglichst gering bleiben.

Koutsoupias (2009) gibt einen guten und aktuellen Überblick über das k-Server-Problem.

Man kann zeigen, dass das Seitenwechselproblem mit Gewichten als Spezialfall des k-Server-Problems aufgefasst werden kann. Für letzteres kann man die folgende untere Schranke zeigen, aus der auch folgt, dass der Reziprok-Algorithmus gegen adaptive Online-Widersacher optimal ist:

11.24 Satz. *Ist R ein randomisierter Online-Algorithmus, der das k-Server-Problem in jedem metrischen Raum löst, dann ist $C_R^{aon} \geqslant k$.*

11.25 Beweis. Es sei R ein randomisierter Online-Algorithmus und H eine Menge von $k+1$ Punkten des Raumes, die zu Beginn (und wie man sehen wird auch danach zu jedem Zeitpunkt) diejenigen k Punkte umfasst, auf denen R seine Server positioniert hat. Wir betrachten im folgenden die Anforderungsfolge $r_1, r_2, \ldots$ bei der jeweils der Punkt aus H als nächstes gewählt wird, auf dem R gerade *keinen* Server positioniert hat. Auf r_1 habe R zu Beginn keinen Server.

Die Kosten von R für die Bedienung von Punkt r_j sind also gerade $d(r_{j+1}, r_j)$. Die Gesamtkosten von R für eine Anforderungsfolge $(r_1, r_2, \ldots, r_n)$ sind also

$$M_R(r_1, \ldots, r_n) = \sum_{j=1}^{n-1} d(r_{j+1}, r_j) + x = \sum_{j=1}^{n-1} d(r_j, r_{j+1}) + x \,,$$

wobei hier x die Kosten für die Bearbeitung von r_n bezeichnet.

Um den Satz zu beweisen, zeigen wir nun, dass es k Online-Algorithmen $B_1, \ldots, B_k$ gibt, deren Kosten $\sum_{i=1}^k M_{B_i}(r_1, \ldots, r_n)$ für die Bearbeitung der Anforderungsfolge *zusammen höchstens* ebenfalls nur $M_R(r_1, \ldots, r_n)$ ist. Also gibt es jedenfalls mindestens ein i, für das die Kosten $M_{B_i}(r_1, \ldots, r_n) \leqslant 1/k \cdot M_R(r_1, \ldots, r_n)$ sind. Dies ist folglich auch eine Schranke für die Kosten des Widersachers.

Es sei $H = \{r_1, u_1, \ldots, u_k\}$. Für $1 \leqslant i \leqslant k$ beginnt Algorithmus B_i so, dass seine Server auf allen Punkten außer u_i positioniert sind. Wann immer B_i keinen Server auf r_j hat, bewegt er den von r_{j-1} nach r_j.

Wir zeigen nun:

1. Es bezeichne S_i die Menge der Punkte, an denen B_i seine Server positioniert hat. Dann gilt: Zu jedem Zeitpunkt ist für $i \neq m$ auch $S_i \neq S_m$.

 © Th. Worsch 2000-2015

2. Bei jeder Anforderung r_j muss nur einer der k Algorithmen einen Server nach r_j bewegen.
3. $\sum_{i=1}^{k} M_{B_i}(r_1, \dots, r_n) \leqslant M_R(r_1, \dots, r_n)$.

zu 1. Die Behauptung wird durch Induktion über die Zeit bewiesen. Bevor die erste Anforderung kommt, gilt die Behauptung nach Konstruktion.

Gelte die Behauptung nun, bevor irgendeine Anforderung r_j kommt. Das heißt dann aber, dass es nicht zwei B_i, B_m geben kann, die keinen Server auf r_j positioniert haben, denn sonst wäre $S_i = S_m$. Also ist entweder $r_j \in S_i \cap S_m$, so dass sich weder S_i noch S_m ändert, sie also verschieden bleiben. Oder es ist etwa $r_j \in S_i \setminus S_m$. Dann bewegt B_m seinen Server von r_{j-1} nach r_j, B_i aber nicht. Also ist hinterher $r_{j-1} \in S_i \setminus S_m$, so dass S_i und S_m wieder verschieden sind.

zu 2. Müssten zwei verschiedene Algorithmen, etwa B_i und B_m für ein r_j einen Server dorthin bewegen, dann wären S_i und S_m gleich, was wie eben gesehen nicht der Fall ist.

zu 3. Da für jedes r_j nur *ein* B_i einen Server dorthin bewegen muss, und zwar von r_{j-1}, liefert r_j tatsächlich nur einen Beitrag von $d(r_{j-1}, r_j)$ zu $\sum_{i=1}^{k} M_{B_i}(r_1, \dots, r_n)$. Dieser Wert ist also $\sum_{j=2}^{n} d(r_{j-1}, r_j) = \sum_{j=1}^{n-1} d(r_j, r_{j+1})$.

Wenn der Widersacher mit gleicher Wahrscheinlichkeit eines der B_i als seine Cacheverwaltungsstrategie wählt, hat er also erwartete Kosten $M_R(r_1, \dots)/k$. ∎

11.26 In der Praxis beobachtet man, dass verschiedene Online-Algorithmen z. B. für Seitenwechsel, die den gleichen Wettbewerbsfaktor haben, trotzdem unterschiedlich gut geeignet sind.

Das liegt salopp gesprochen daran, dass in der Realität auftretende Anforderungsfolgen nicht so willkürlich sind, wie sie ein Widersacher wählt. Es gibt mehrere Ansätze, dies zu modellieren. Karlin, S. J. Phillips und Raghavan (2000) verfolgen den Anzsatz, die Anforderungsfolgen von einer Markov-Kette „erzeugen zu lassen". Ein verwandter, aber anderer Ansatz ist die Verwendung von *access graphs*; hierzu lese man die Arbeiten von Allan Borodin, Irani u. a. (1995) und Irani, Karlin und S. Phillips (1996). Albers, Favrholdt und Giel (2005) berücksichtigen bei ihrer Analyse die Tatsache, dass aufeinander folgende Speicherzugriffe üblicherweise eine gewisse *Lokalität* aufweisen.

Wir beschließen dieses Kapitel mit einigen allgemeinen Ergebnissen zu adaptiven Widersachern, ohne noch auf die Beweise einzugehen. Daran Interessierte seien auf das Buch von Motwani und Raghavan (1995) und die dort angegebenen Literaturstellen verwiesen.

11.27 Die im folgenden auftretenden Konstanten C^{xyz} seien definiert als die Infima der C_R^{xyz}, genommen über alle randomisierten Algorithmen R für das Seitenwechselproblem.

Zunächst einmal ist aufgrund der Definitionen klar:

$$C^{obl} \leqslant C^{aon} \leqslant C^{aof} \leqslant C^{det} .$$

Weniger klar ist:

$$C^{aon} \geqslant \frac{C^{det}}{C^{obl}} = \Omega(k/\ln k) .$$

Das kann man folgern aus:

11.28 Satz. *Wenn R ein randomisierter Algorithmus ist, der α-kompetitiv gegen adaptive Online-Widersacher ist, und wenn es einen β-kompetitiven randomisierten Algorithmus gegen unwissende Widersacher gibt, dann ist R auch $\alpha\beta$-kompetitiv gegen adaptive Offline-Widersacher.*

 © Th. Worsch 2000-2015

Der letzte Satz kann so aufgefasst werden, dass adaptive Offline-Widersacher sehr stark sind. Gegen sie hilft Randomisierung nicht:

11.29 Satz. *Wenn es einen randomisierten Algorithmus gibt, der α-kompetitiv gegen jeden adaptiven Offline-Widersacher ist, dann gibt es auch einen deterministischen Algorithmus, der α-kompetitiv ist.*

Das erklärt dann auch, warum man bei randomisierten Algorithmen überhaupt die anderen Widersacher betrachtet: Andernfalls bringt Randomisierung nämlich gar nichts.

Zusammenfassung

1. Zur Beurteilung der Qualität von Onlinealgorithmen für das Seitenwechselproblem (und anderen) ist eine Worst-Case-Analyse sinnlos. Stattdessen ist die Betrachtung von Widersachern hilfreich.
2. Für kompetitive Analyse sind manchmal geschickt gewählte Potenzialfunktionen hilfreich.
3. Gegen unwissende Widersacher erreicht man randomisiert Wettbewerbsfaktoren in $O(\log n)$ während deterministisch $O(n)$ das Optimum ist.

Literatur

Albers, Susanne, Lene M. Favrholdt und Oliver Giel (2005). „On paging with locality of reference". In: *Journal of Computer and System Sciences* 70.2, S. 145–175 (siehe S. 102).

Belady, L. A. (1966). „A study of replacement algorithms for virtual storage computers". In: *IBM Systems Journal* 5.2, S. 78–101 (siehe S. 90).

Borodin, Allan, Sandy Irani u. a. (1995). „Competitive Paging with Locality of Reference". In: *Journal of Computer and System Sciences* 50.2, S. 244–258 (siehe S. 102).

Borodin, Allan und Ran El-Yaniv (1998). *Online Computation and Competitive Analysis*. Cambridge Unviersity Press. ISBN: 0-521-56392-5 (siehe S. 8, 90).

Fiat, A. u. a. (1991). „Competitive Paging Algorithms". In: *Journal of Algorithms* 12.4, S. 685–699 (siehe S. 93, 95).

Irani, Sandy, Anna R. Karlin und Steven Phillips (1996). „Strongly Competitive Algorithms for Paging with Locality of Reference". In: *SIAM Journal on Computing* 25.3, S. 477–497 (siehe S. 102).

Karlin, Anna R., Steven J. Phillips und Prabhakar Raghavan (2000). „Markov Paging". In: *SIAM Journal on Computing* 30.3, S. 906–922 (siehe S. 102).

Koutsoupias, Elias (2009). „The k-server problem". In: *Computer Science Review* 3.2, S. 105–118 (siehe S. 101).

Mattison, R. L. u. a. (1971). „Evaluation techniques for storage hierarchies". In: *IBM Systems Journal* 9.2 (siehe S. 90).

McGeoch, Lyle A. und Daniel Dominic Sleator (1991). „A Strongly Competitive Randomized Paging Algorithm". In: *Algorithmica* 6, S. 816–825 (siehe S. 96).

Motwani, Rajeev und Prabhakar Raghavan (1995). *Randomized Algorithms*. Cambridge University Press. ISBN: 0-521-47465-5 (siehe S. 8, 96, 102).

 © Th. Worsch 2000-2015

Sleator, Daniel D. und Robert E. Tarjan (1985). „Amortized Efficiency of List Update and Paging Rules“. In: *Communications of the ACM* 28, S. 202–208 (siehe S. 91).

Tarjan, Robert Endre (1985). „Amortized Computational Complexity“. In: *SIAM Journal Alg. Disc. Meth.* 6.2, S. 306–318 (siehe S. 97).

Ungerer, Theo (1995). *Mikroprozessortechnik*. International Thomson Publishing (siehe S. 91).

 © Th. Worsch 2000-2015

12 Hashing

12.1 Grundlagen

Gegeben ist ein Universum M der Kardinalität $|M| = m$. Davon soll eine Untermenge $S \subseteq M$ von Schlüsseln verwaltet werden. Dies soll mit Hilfe einer Hashtabelle T geschehen. Ihre Einträge werden mit Indizes aus einer Menge N mit $|N| = n < m$ adressiert. Dazu wird eine Hashfunktion $h : M \to N$ benutzt. Die Idee besteht darin, die zu einem Schlüssel $s \in S$ gehörende Information über den Eintrag $T[h(s)] = (s, \ldots)$ zugänglich zu machen.

12.1 Definition Eine Hashfunktion heißt *perfekt* für eine Schlüsselmenge S, falls für alle $x, y \in S$ gilt: $x \neq y \implies h(x) \neq h(y)$. ◇

Ist $|S| > |N|$, dann muss es nach dem Schubfachprinzip zu einer *Kollision* kommen, d. h. es existieren verschiedene Schlüssel x und y mit $h(x) = h(y)$ und eine perfekte Hashfunktion kann in diesem Fall nicht existieren.

Ist $|M| > |N|$, dann kann es keine Hashfunktion geben, die für alle Schlüsselmengen perfekt ist.

Zur Auflösung von Kollisionen gibt es verschiedene Strategien. Eine Methode besteht darin, jeweils alle Schlüssel mit gleichem Hashwert in einer verketteten Liste zu speichern. Eine Alternative ist es, solche Schlüssel mit einer weiteren Hashfunktion auf jeweils eine (kleinere) zweite Hashtabelle abzubilden.

Es gibt mehrere Anwendungen von Hashfunktionen im Bereich randomisierter Algorithmen. Ausgangspunkt ist im allgemeinen eine ganze sogenannte universelle Familie H von Hashfunktionen. Das ist Thema der Abschnitte 12.2 und 12.3.

12.2 Beim Entwurf randomisierter Algorithmen wird mitunter aus H zufällig eine Hashfunktion ausgewählt und benutzt. Das werden wir beim dynamischen und beim statischen Wörterbuchproblem in den Abschnitten 12.4 bzw. 12.5 genauer sehen.

Eine weitere Anwendung für universelle Familien von Hashfunktionen ist die Einsparung von Zufallsbits bei randomisierten Algorithmen. Darauf werden wir in Abschnitt 12.6 zu sprechen kommen.

12.2 Universelle Familien von Hashfunktionen

12.3 Definition Es seien M und N wie oben mit $|M| = m \geqslant |N| = n$. Eine Familie H von Hashfunktionen $h : M \to N$ heißt 2-*universell*, wenn für alle $x, y \in M$ mit $x \neq y$ gilt: Für ein zufällig gleichverteilt aus H ausgewähltes h ist $\mathbf{Pr}\left[h(x) = h(y)\right] \leqslant 1/n$. ◇

Man beachte, dass $1/n$ auch die Wahrscheinlichkeit ist, dass $h(x)$ und $h(y)$ gleich sind, wenn man sie unabhängig zufällig gleichverteilt wählt.

12.4 Ein einfaches Beispiel einer 2-universellen Familie ist die Menge *aller* Hashfunktionen $H = N^M$. In diesem Fall ist $\mathbf{Pr}[h(x) = h(y)] = 1/n$, denn zu jedem h mit $h(x) = h(y)$ gibt es $n-1$ andere Hashfunktionen, die sich von h nur an der Stelle y unterscheiden.

Diese Familie von Hashfunktionen hat aber für Anwendungen noch zwei Nachteile. Zum einen benötigt man $\Theta(m \log n)$ Zufallsbits, um ein $h \in H$ auszuwählen, zum anderen ist nicht sichergestellt, dass man jedes h „leicht" berechnen kann.

12.5 Definition Mit den Bezeichnungen wie bisher und für $X, Y \subseteq M$ definieren wir:

$$\begin{aligned}
\delta(x,y,h) &= \begin{cases} 1 & \text{falls } x \neq y \wedge h(x) = h(y) \\ 0 & \text{sonst} \end{cases} \\
\delta(x,y,H) &= \sum_{h \in H} \delta(x,y,h) \\
\delta(x,Y,h) &= \sum_{y \in Y} \delta(x,y,h) \\
\delta(X,Y,h) &= \sum_{x \in X} \delta(x,Y,h) \\
\delta(X,Y,H) &= \sum_{h \in H} \delta(X,Y,h)
\end{aligned}$$

◇

12.6 Ist H eine 2-universelle Familie, so ist für $x \neq y$ stets $\delta(x,y,H) \leqslant |H|/n$. Denn andernfalls müsste bei zufälliger gleichverteilter Wahl eines $h \in H$ von $|H|$ vielen offensichtlich $\mathbf{Pr}[h(x) = h(y)] > 1/n$ sein.

Die Abschätzung $\delta(x,y,H) \leqslant |H|/n$ für eine 2-universelle Familie ist recht gut, wie das folgende Ergebnis zeigt.

12.7 Satz. *Für jede Familie H von Funktionen $h : M \to N$ existieren x und y aus M, so dass $\delta(x,y,H) \geqslant |H|/n - |H|/m$.*

Vor dem Beweis sei noch die folgende kurze Rechnung eingeschoben.

12.8 Es seien k und g zwei positive ganze Zahlen und $A(k,g) = k(k-1) + g(g-1)$. Für $k \leqslant g-1$ gilt dann $A(k,g) \geqslant A(k+1, g-1)$, denn es ist

$$\begin{aligned}
A(k,g) - A(k+1,g-1) &= k(k-1) + g(g-1) - \big((k+1)k + (g-1)(g-2)\big) \\
&= k^2 - k + g^2 - g - \big(k^2 + k + g^2 - 3g + 2\big) \\
&= -k - g - k + 3g - 2 \\
&= 2(g-1-k) \\
&\geqslant 0\,.
\end{aligned}$$

12.9 Beweis. (von Satz 12.7) Es sei $h \in H$ beliebig. Für $z \in N$ sei $A_z = \{x \in M \mid h(x) = z\}$.
Für $w, z \in N$ gilt:

$$\delta(A_w, A_z, h) = \begin{cases} 0 & \text{falls } w \neq z \\ |A_z|\,(|A_z| - 1) & \text{falls } w = z \end{cases},$$

 © Th. Worsch 2000-2015

da alle Paare (x, y) mit $x \neq y$ genau 1 zur Summe beitragen.

Wegen der Rechnung in Punkt 12.8 wird die Gesamtzahl der Kollisionen minimiert, wenn die Mengen A_z gleich groß sind. Also ist

$$\delta(M, M, h) = \sum_{z \in N} |A_z| (|A_z| - 1) \geqslant n \frac{m}{n} \left(\frac{m}{n} - 1 \right) = m^2 \left(\frac{1}{n} - \frac{1}{m} \right) .$$

Folglich ist $\delta(M, M, H) \geqslant |H| \, m^2 (1/n - 1/m)$. Daher existieren nach dem Schubfachprinzip $x, y \in M$ mit $\delta(x, y, H) \geqslant \delta(M, M, H)/m^2 \geqslant |H| (1/n - 1/m)$. ■

12.3 Zwei 2-universelle Familien von Hashfunktionen

O. B. d. A. sei nun $M = \{0, \dots, m-1\}$ und $N = \{0, \dots, n-1\}$.

In der weiter unten beschriebenen Konstruktion wird eine Primzahl $p \geqslant m$ benötigt. Der folgende Satz, dessen Aussage auch als „Bertrands Postulat“ bezeichnet wird, stellt sicher, dass es stets eine relativ kleine solche Zahl gibt.

12.10 Satz. (Chebyshev) *Zu jedem $m > 1$ gibt es eine Primzahl p mit $m \leqslant p < 2m$.*

12.11 Definition Es seien $m \geqslant n$ und $p \geqslant m$ eine Primzahl. Die Menge $H = \{h_{a,b} \mid a, b \in \mathbb{Z}_p \wedge a \neq 0\}$ von Hashfunktionen ist wie folgt definiert:

$$\begin{aligned} h_{a,b}(x) &= g(f_{a,b}(x)) \\ \text{mit} \quad f_{a,b} : \mathbb{Z}_p &\to \mathbb{Z}_p \\ x &\mapsto ax + b \bmod p \\ g : \mathbb{Z}_p &\to N \\ x &\mapsto x \bmod n \end{aligned}$$

Benutzt wird beim Hashen eigentlich nur die Einschränkung der $h_{a,b} : \mathbb{Z}_p \to N$ auf $M \subseteq \mathbb{Z}_p$. Hierdurch kann die Zahl der Kollisionen offensichtlich nicht größer werden. ◇

Ziel ist nun der Nachweis des folgenden Satzes.

12.12 Satz. *Die Familie H aus Definition 12.11 ist 2-universell.*

12.13 Beweis. Zu zeigen ist, dass bei zufällig gewähltem $h_{a,b} \in H$ gilt: $\mathbf{Pr}\left[h_{a,b}(x) = h_{a,b}(y)\right] \leqslant 1/n$. Dies kann auch formuliert werden als $\delta(x, y, H) \leqslant |H| / n$.

Der Beweis wird in zwei Schritten geführt. Zunächst wird gezeigt, dass für $x \neq y$ gilt: $\delta(x, y, H) = \delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$. In einem zweiten Schritt wird $\delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$ nach oben abgeschätzt.

1. Es seien x und y so, dass es zu einer Kollision $h_{a,b}(x) = h_{a,b}(y)$ für ein $h_{a,b}$ kommt. Es ist also $g(f_{a,b}(x)) = g(f_{a,b}(y))$.

   Da $a \neq 0$ und p prim ist, sind alle $f_{a,b}$ bijektiv. Also ist $r = f_{a,b}(x) \neq f_{a,b}(y) = s$. Eine Kollision passiert also immer genau dann, wenn $g(r) = g(s)$, d. h. $r \equiv s \mod n$.

© Th. Worsch 2000-2015

Umgekehrt legt die Wahl von Werten $x \neq y$ und $r \neq s$ *eindeutig* genau eine Hashfunktion $h_{a,b}$ fest, so dass $f_{a,b}(x) = r$ und $f_{a,b}(y) = s$, denn es muss das lineare Gleichungssystem

$$\begin{aligned} ax + b &\equiv r \pmod p \\ ay + b &\equiv s \pmod p \end{aligned}$$

über $\mathbb{Z}_p$ erfüllt werden.

Also ist die Anzahl $\delta(x, y, H)$ der Hashfunktionen, für die x und y zu einer Kollision führen, gleich der Anzahl $\delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$ von Paaren (r, s) mit $r \neq s$ und $r \equiv s \mod n$.

2. Damit bleibt noch abzuschätzen, wieviele solche Paare es höchstens gibt.

   Für $z \in \mathbb{N}$ sei dazu $A_z = \{x \in \mathbb{Z}_p \mid g(x) = z\}$. Für jedes z ist $|A_z| \leqslant \lceil p/n \rceil$. Folglich gibt es zu jedem r höchstens $\lceil p/n \rceil$ Werte s, so dass r und s alle Bedingungen erfüllen. Also ist (wegen der Forderung $r \neq s$)

$$\delta(\mathbb{Z}_p, \mathbb{Z}_p, g) \leqslant p\left(\left\lceil \frac{p}{n} \right\rceil - 1\right) \leqslant \frac{p(p-1)}{n} = \frac{|H|}{n} .$$

■

Wegen der großen Bedeutung von (universellen Familien von) Hashfunktionen gehen wir noch auf eine zweite ein.

12.14 Definition Es sei nun n eine Primzahl. (Falls das zunächst nicht der Fall ist, vergrößere man die Hashtabelle. Satz 12.10 sichert zu, dass man dafür nicht „nicht allzu viel" zusätzlichen Speicherplatz benötigt.

Ein Schlüssel x werde dargestellt als Folge $\langle x_0, x_1, \ldots, x_r \rangle$ von $r+1$ Werten x_i, für die alle gilt: $0 \leqslant x_i \leqslant n-1$. Analog beschreiben wir Hashfunktionen h_a durch eine Folge $a = \langle a_0, a_1, \ldots, a_r \rangle$ von $r+1$ Werten a_i mit $0 \leqslant a_i \leqslant n-1$. Es sei

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod n .$$

Die interessierende Familie von Hashfunktionen ist die Menge aller n^{r+1} solchen h_a. ◇

12.15 Satz. *Die in Definition 12.14 festgelegte Familie von Hashfunktionen ist 2-universell.*

12.16 Beweis. Es sei $x \neq y$ und etwa $x_0 \neq y_0$. (Der Fall $x_i \neq y_i$ für ein $i > 0$ kann analog behandelt werden.)

Es seien $a_1, \ldots, a_r$ beliebig aber fest. Es ist

$$a_0(x_0 - y_0) = -\sum_{i=1}^{r} a_i(x_i - y_i) \bmod n .$$

Da n prim ist, ist $x_0 - y_0$ invertierbar modulo n und daher a_0 eindeutig bestimmt. Also kollidieren zwei Werte x und y für genau n^r der n^{r+1} möglichen a bzw. h_a. Also ist die Wahrscheinlichkeit für eine Kollision gerade $1/n$. ■

© Th. Worsch 2000-2015

12.4 Das dynamische Wörterbuchproblem

12.17 Bei den Wörterbuchproblemen besteht die Aufgabe darin, für eine Reihe von „Anfragen“ der Form FIND(x) jeweils festzustellen, ob x in einer Menge von Schlüsseln S enthalten ist. Man betrachtet üblicherweise zwei Problemvarianten:

1. das *dynamische Wörterbuchproblem*: Hier liegt S nicht von vorneherein fest, sondern ergibt sich durch Operationen INSERT(x) und DELETE(x), die zwischen die FIND-Operationen eingestreut sind.

2. das *statische Wörterbuchproblem*: Die Menge S ist hier von vorneherein festgelegt. Die Aufgabe besteht darin, eine Folge von Anfragen FIND(x_1), FIND(x_2), usw. zu bearbeiten und jedes Mal festzustellen, ob x_i in der Hashtabelle eingetragen ist.

Die grundsätzliche Idee für randomisierte Algorithmen zur „Wörterbuchverwaltung“ besteht darin, aus einer Familie H von Hashfunktionen zu Beginn zufällig ein h auszuwählen und damit zu arbeiten. Bei geeigneter Wahl von H sind nicht nur die $h \in H$ leicht zu berechnen und mit wenigen Zufallsbits auszuwählen, sondern die Wahrscheinlichkeit für Kollisionen kann auch auf ein annehmbares Maß reduziert werden.

Wir wollen zunächst die folgende naheliegende Vorgehensweise für das dynamische Wörterbuchproblem analysieren.

12.18 ALGORITHMUS. Zu Beginn wird aus einer 2-universellen Familie H zufällig gleichverteilt eine Hashfunktion h ausgewählt und dann bei der Abarbeitung einer (vorher unbekannten) Liste $R = R_1, R_2, \ldots, R_r$ von Operationen benutzt.

Bei einer INSERT-Operation für Schlüssel x wird er in der Hashtabelle an Stelle $h(x)$ eingefügt. Im Falle einer Kollision wird eine lineare verkette Liste aufgebaut.

Bei einer FIND-Operation wird in der Hashtabelle an Stelle $h(x)$ (bzw. ggf. in der dort angehängten linearen Liste) nach x gesucht.

12.19 Ist zu einem Zeitpunkt S die Menge der gerade gespeicherten Schlüssel und soll ein weiterer Schlüssel x eingefügt werden, so ist $\delta(x, S, h)$ die Länge der lineare Liste, in die x eingefügt wird.

Den Erwartungswert für ihre Länge wollen wir nun abschätzen.

12.20 LEMMA. Für alle $x \in M$ und $S \subseteq M$ und ein zufällig gewähltes h aus einer 2-universellen Familie H gilt:

$$\mathbf{E}\left[\delta(x, S, h)\right] \leqslant \frac{|S|}{n} .$$

12.21 BEWEIS.

$$\mathbf{E}\left[\delta(x, S, h)\right] = \sum_{h \in H} \frac{\delta(x, S, h)}{|H|} = \frac{1}{|H|} \sum_{h \in H} \sum_{y \in S} \delta(x, y, h) = \frac{1}{|H|} \sum_{y \in S} \delta(x, y, H) \leqslant \frac{1}{|H|} \sum_{y \in S} \frac{|H|}{n} = \frac{|S|}{n} .$$

Dabei rührt das „$\leqslant$“ von der Überlegung in Punkt 12.6 her. ∎

12.22 SATZ. *Mit den Bezeichnungen wie in Algorithmus 12.18 ist der Erwartungswert für den Gesamtzeitaufwand $t(R, h)$ zur Abarbeitung einer Liste R mit s INSERT-Operationen bei zufällig gewähltem $h \in H$*

$$\mathbf{E}\left[t(R, h)\right] \leqslant r\left(1 + \frac{s}{n}\right) .$$

© Th. Worsch 2000-2015

12.23 BEWEIS. $\mathbf{E}[t(R,h)]$ ist kleiner oder gleich dem r-fachen des maximal zu erwartenden Zeitbedarfs für die Bearbeitung einer Operation. Dieser ist im wesentlichen beschränkt durch den Erwartungswert für die Länge der jeweils zu untersuchenden linearen Liste, der nach Punkt 12.20 maximal s/n ist. ■

12.24 Der Erwartungswert für den Zeitaufwand pro Operation ist also kleiner oder gleich $1 + s/n$. Kann man sich erlauben, $n \geqslant s$ zu wählen, dann ist der Erwartungswert durch die Konstante 2 beschränkt.

Allerdings kann im schlimmsten Fall immer noch ein Zeitaufwand proportional zu s pro FIND-Operation auftreten.

12.5 Das statische Wörterbuchproblem

Im folgenden betrachten wir das statische Wörterbuchproblem. Die zu verwaltende Schlüsselmenge S liegt also von vorne herein fest. Das wird ausgenutzt werden, um vor der Beantwortung von Suchanfragen eine gewisse „Vorverarbeitung“ vorzunehmen, in deren Verlauf die zu verwendende Hashfunktion bestimmt wird.

Wir benutzen wieder die universelle Familie H von Hashfunktionen $h_{a,b}(x) = ax + b$ aus Definition 12.11.

12.25 Aus der Universalität von H folgt, dass bei zufälliger Wahl einer Hashfunktion h der Erwartungswert für die Anzahl von Kollisionen für eine Schlüsselmenge $|S|$ gerade

$$\sum_{x \neq y \in S} \mathbf{Pr}[h(x) = h(y)] \leqslant \binom{|S|}{2} \cdot \frac{1}{n}$$

ist. Wählt man also z. B. eine Tabelle der Größe $n = |S|^2$, so ist der Erwartungswert kleiner oder gleich 1/2. Das heißt aber auch, dass mit einer Wahrscheinlichkeit größer oder gleich 1/2 jenes h injektiv ist.

Wählt man eine Tabelle der Größe $n = |S|$, so ist der Erwartungswert kleiner oder gleich $|S|/2$. Die Wahrscheinlichkeit, dass für ein zufällig gewähltes h die Anzahl der Kollisionen größer oder gleich $|S|$ ist, ist daher nach der Markov-Ungleichung (Satz 4.10) kleiner oder gleich 1/2.

12.26 Kann man sich eine Hashtabelle der Größe $|S|^2$ leisten, dann kann man alle Anfragen schnell beantworten, indem man in der Vorverarbeitungsphase eine zufällig gewählte Hashfunktion h nach der anderen durch vollständige Überprüfung aller Funktionswerte daraufhin untersucht, ob h injektiv ist. Solche h müssen nach Punkt 12.25 existieren und man findet mit großer Wahrscheinlichkeit auch schnell eines. (Der Zeitbedarf für die Überprüfung jedes h ist in $\Theta(|S|)$.)

Im weiteren soll nun noch der Fall betrachtet werden, dass die Hashtabelle nur eine Größe proportional zu $|S|$ hat. Ziel ist auch in diesem Fall eine Datenstruktur, die jede FIND-Operation in konstanter Zeit erledigen kann, wiederum um den Preis einer Vorverarbeitung. Sie wird aber wie oben nur Zeitbedarf $\Theta(|S|)$ haben.

12.27 (DATENSTRUKTUREN FÜR ALGORITHMUS 12.28) Die im folgenden dargestellte Idee, Hashtabellen zweistufig zu benutzen, stammt von Fredman, Komlós und Szemerédi (1984).

 © Th. Worsch 2000-2015

Es gibt eine primäre Hashtabelle N. Für jedes $i \in N$ gibt es eine sekundäre Hashtabelle N_i mit einer separat zu bestimmenden Hashfunktion h_i. In $N[i]$ werden die Parameter gespeichert, die h_i charakterisieren und jeweils einen Verweis auf eine sekundäre Hashtabelle N_i. Deren Größe wird im nachfolgenden Algorithmus so gewählt, dass der Gesamtspeicherbedarf in $\Theta(n)$, also proportional zur Anzahl Schlüssel, bleibt.

12.28 Algorithmus.

$N \leftarrow$ ⟨*Hashtabelle der Größe* $n = |S|$⟩
$H \leftarrow$ ⟨*Hashfamilie für* N⟩

⟨*1. Phase: suche primäre Hashfunktion*⟩

```
repeat
    h ← ⟨zufällig aus H gewählt⟩
    for i ← 0 to n − 1 do
        S_i ← ∅
        k_i ← 0
    od
    for each s ∈ S do
        ⟨add s to S_h(s)⟩
        k_h(s) ← k_h(s) + 1
    od
until ∑_{i<n} (k_i über 2) < n        ⟨Anzahl Kollisionen kleiner n⟩
```

⟨*2. Phase: suche sekundäre injektive Hashfunktionen*⟩

```
for i ← 0 to n − 1 do
    N_i ← ⟨Hashtabelle der Größe k_i^2⟩
    H_i ← ⟨Hashfamilie für Tabelle mit k_i^2 Einträgen⟩
    repeat
        h_i ← ⟨zufällig aus H_i gewählt⟩
    until ⟨h_i ist auf S_i injektiv⟩
od
```

12.29 (Analyse von Algorithmus 12.28)

1. Wegen der letzten Bemerkung in Punkt 12.25 ist der Erwartungswert für die Anzahl der Durchläufe durch die **repeat**-Schleife in der 1. Phase höchstens 2 und somit der Erwartungswert für deren Zeitbedarf in $O(n)$.

2. Analog ist in der 2. Phase wegen der vorletzten Bemerkung in Punkt 12.25 der Erwartungswert für die Anzahl der Durchläufe durch die **repeat**-Schleife jeweils konstant und daher der Zeitbedarf (der durch den für den Test nach **until** bestimmt wird) proportional zu k_i^2.

   Der Gesamtzeitbedarf für die 2. Phase daher größenordnungsmäßig durch $\sum \binom{k_i}{2}$ bestimmt, und diese Größe ist durch die Gesamtzahl Kollisionen, also durch n nach oben beschränkt.

© Th. Worsch 2000-2015

12.6 Derandomisierung

Eine weitere schöne Anwendung von Hashfunktionen ist die *Derandomisierung* randomisierter Algorithmen; damit ist die Verringerung der Anzahl benötigter Zufallsbits gemeint, ohne die erwartete Laufzeit gravierend zu erhöhen. Die totale Elimination von Zufall ist auf diese Weise allerdings bislang nicht generell möglich (obwohl manche Forscher vermuten, dass **P** = **BPP** ist). Für eine vertiefende Darstellung dieses Themas sei auf die Mitschriften einer Vorlesung von Goldreich (2001) und auf den technischen Bericht von Luby und Wigderson (1995) verwiesen.

12.30 Die grundsätzliche Idee wird darin bestehen, aus einer Zahl echter Zufallsbits mit Hilfe von Hashfunktionen mehr Bits zu berechnen, die zwar nicht mehr echt unabhängig voneinander sind, „aber noch so aussehen“, d. h. man weiß über diese Bits noch so viel, dass man sie in Rechnungen/Abschätzungen vorteilhaft einsetzen kann.

12.31 Definition Zufallsvariablen $X_1, \dots, X_k$ mit gleichen Definitionsbereich heißen *paarweise unabhängig*, wenn für alle $i \neq j$ und alle x und y gilt:

$$\mathbf{Pr}\left[X_i = x \wedge X_j = y\right] = \mathbf{Pr}\left[X_i = x\right] \cdot \mathbf{Pr}\left[X_j = y\right] \ .$$

In diesem Fall ist $\mathbf{E}\left[X_i X_j\right] = \mathbf{E}\left[X_i\right] \mathbf{E}\left[X_j\right]$. ◇

12.32 Bemerkung. Man beachte, dass paarweise Unabhängigkeit eine schwächere Forderung ist als (totale) Unabhängigkeit. Sind etwa X und Y unabhängige Zufallsvariablen mit Wertebereich $\{0, 1\}$ und ist Z eine dritte mit $Z = X \oplus Y$, dann sind X, Y, Z paarweise unabhängig, aber (offensichtlich) nicht unabhängig.

In diesem Abschnitt werden wir uns (technisch gesehen) für die Summe paarweise unabhängiger Zufallsvariablen interessieren. Da wir die Ungleichung von Chebyshev anwenden wollen, benötigen wir Informationen über die Varianz solcher Summen.

12.33 Lemma. Es seien $X_1, \dots, X_k$ paarweise unabhängige Zufallsvariablen. Dann ist

$$\mathbf{var}\left[\sum_i X_i\right] = \sum_i \mathbf{var}\left[X_i\right] \ .$$

12.34 Beweis. Da die Varianz einer Zufallsvariablen $\mathbf{var}\left[X\right] = \mathbf{E}\left[X^2\right] - (\mathbf{E}\left[X\right])^2$ ist, ergibt sich:

$$\begin{aligned}
\mathbf{var}\left[\sum_i X_i\right] &= \mathbf{E}\left[\left(\sum_i X_i\right)^2\right] - \left(\mathbf{E}\left[\sum_i X_i\right]\right)^2 \\
&= \mathbf{E}\left[\sum_i X_i^2 + \sum_{i \neq j} X_i X_j\right] - \left(\sum_i \mathbf{E}\left[X_i\right]\right)^2 \\
&= \sum_i \mathbf{E}\left[X_i^2\right] + \sum_{i \neq j} \mathbf{E}\left[X_i X_j\right] - \sum_i (\mathbf{E}\left[X_i\right])^2 - \sum_{i \neq j} \mathbf{E}\left[X_i\right] \mathbf{E}\left[X_j\right] \\
&= \sum_i \mathbf{E}\left[X_i^2\right] - \sum_i (\mathbf{E}\left[X_i\right])^2 + \sum_{i \neq j} \underbrace{\mathbf{E}\left[X_i X_j\right] - \mathbf{E}\left[X_i\right] \mathbf{E}\left[X_j\right]}_{=0} \\
&= \sum_i \left(\mathbf{E}\left[X_i^2\right] - (\mathbf{E}\left[X_i\right])^2\right) \\
&= \sum_i \mathbf{var}\left[X_i\right]
\end{aligned}$$

 © Th. Worsch 2000-2015

■

12.35 Definition Es seien M und N wie oben mit $|M| = m \geqslant |N| = n$. Eine Familie H von Hashfunktionen $h : M \to N$ heißt *stark 2-universell*, wenn für alle $x, y \in M$ mit $x \neq y$ und für alle $x', y' \in N$ gilt: Für ein zufällig gleichverteilt aus H ausgewähltes h ist $\mathbf{Pr}[h(x) = x' \wedge h(y) = y'] = 1/n^2$. ◇

12.36 Offensichtlich sind stark 2-universelle Familie von Hashfunktionen 2-universell im Sinne von Definition 12.3, denn $\mathbf{Pr}[h(x) = h(y)] = \sum_{x' \in N} \mathbf{Pr}[h(x) = x' \wedge h(y) = x'] = |N|/n^2 = 1/n$.

Für jedes $x' \in N$ gilt außerdem:

$$\mathbf{Pr}\left[h(x) = x'\right] = \sum_{y' \in N} \mathbf{Pr}\left[h(x) = x' \wedge h(y) = y'\right] = 1/n$$

Also ist für eine zufällig gleichverteilt gewählte Hashfunktion aus einer stark 2-universellen Familie

$$\mathbf{Pr}\left[h(x) = x' \wedge h(y) = y'\right] = \mathbf{Pr}\left[h(x) = x'\right] \cdot \mathbf{Pr}\left[h(y) = y'\right]$$

12.37 Beispiel. Eine leichte Abwandlung der Familie aus Definition 12.11 liefert eine stark 2-universelle Familie von Hashfunktionen. Wir gehen im folgenden davon aus, dass $m = n = 2^r$ ist. Wörter, die r Bits lang sind, interpretieren wir als Elemente des Körpers $F = GF[2^r]$. Für $a, b \in F$ sei $h_{a,b} : F \to F$ die Hashfunktion mit $h_{a,b}(i) = ai + b$. Die Menge der 2^{2r} solchen Hashfunktionen ist eine stark 2-universelle Familie.

12.38 Für den Rest dieses Abschnittes benutzen wir folgende Bezeichnungen. Wahrscheinlichkeitsraum Ω ist eine stark 2-universelle Familie von Hashfunktionen (die von M nach N abbbilden). Für $i \in M$ sei H_i die Zufallsvariable auf Ω mit $H_i(h) = h(i)$. Die Ausführungen in Punkt 12.36 zeigen, dass diese Zufallsvariablen H_i paarweise unabhängig sind.

Wir führen nun noch ein wenig Notation ein.

12.39 Es sei $L \subseteq A^*$ eine formale Sprache und T eine randomisierte **BPP**- oder **RP**-Turingmaschine für L. Wir fassen im folgenden T so auf, dass es ein „echtes“ Wort $x \in A^n$ und eine passende Folge $y \in \{0,1\}^r$ von Zufallsbits als zusätzliche Eingabe erhält. Wir schreiben $T(x, y) \in \{0, 1\}$ für das Ergebnis der Berechnung. Mit $W_x = \{y \mid T(x, y) = 1\}$ bezeichnen wir die Menge der y, die bezeugen, dass $x \in L$ ist. Außerdem sei $\mu(W_x) = |W_x|/2^r$.

Im Fall von **RP** gilt also

$$\begin{aligned} x \in L &\Rightarrow \mu(W_x) > 1/2 \\ x \notin L &\Rightarrow \mu(W_x) = 0 \end{aligned}$$

und im Fall von **BPP** gilt

$$\begin{aligned} x \in L &\Rightarrow \mu(W_x) > 3/4 \\ x \notin L &\Rightarrow \mu(W_x) < 1/4 \end{aligned}$$

Außerdem ist in beiden Fällen r polynomiell in n.

12.40 Definition Eine formale Sprache $L \subseteq A^*$ ist in **P**/**poly**, wenn es eine deterministische Polynomialzeit-Turingmaschine $T(x, y)$ und ein Polynom $r(n)$ gibt mit der Eigenschaft:

$$\forall n\ \exists y \in \{0,1\}^{r(n)}\ \forall x \in A^n\ : \begin{cases} x \in L \Rightarrow T(x,y) = 1 \\ x \notin L \Rightarrow T(x,y) = 0 \end{cases}$$ ◇

 © Th. Worsch 2000-2015

12.41 SATZ.

$$\mathbf{RP} \subseteq \mathbf{P/poly} \qquad \textit{und} \qquad \mathbf{BPP} \subseteq \mathbf{P/poly}$$

Das bedeutet, dass es für jede Wortlänge n *ein einzelnes Wort* y gibt, das für alle Eingaben der Länge n so viel „Hilfestellung“ gibt, dass man deterministisch in Polynomialzeit das Wortproblem entscheiden kann. Könnte man diese „advice strings“ in Polynomialzeit berechnen, wäre $\mathbf{P/poly} = \mathbf{P}$. Das kann man aber eben im allgemeinen nicht.

Wir beschränken uns im folgenden auf den Beweis für den Fall **RP**, der von L. Adleman (1978) stammt.

12.42 BEWEIS. Sei etwa T eine **RP**-Turingmaschine, die eine Sprache L akzeptiert. Es sei n eine Wortlänge und $r = r(n)$ für ein Polynom $r(n)$. Man betrachte die Matrix M, von der man sich vorstelle, dass ihre Zeilen mit den $|A|^n$ Wörtern $x \in A^n$ indiziert werden und ihre Spalten mit allen 2^r möglichen Folgen von r (Zufalls-)Bits. Der Eintrag M_{xy} sei 1 oder 0, je nach dem, ob $y \in W_x$ ist oder nicht.

Als erstes entferne man die Zeilen von M, die zu Wörtern gehören, die nicht in L sind. Die entstehende Matrix heiße M_0. In jeder ihrer Zeilen ist mindestens die Hälfte aller Einträge 1.

Wenn in einer Matrix in jeder Zeile mindestens die Hälfte aller Einträge 1 ist, dann ist insgesamt mindestens die Hälfte aller Einträge 1, und dann muss in mindestens einer Spalte mindestens die Hälfte aller Einträge 1 sein. Sei y_0 der Index dieser Spalte.

Zu gegebenen M_i und y_i sei allgemein M_{i+1} die Matrix, die aus M_i entsteht, wenn man Spalte y_i streicht und auch alle Zeilen, die in der gestrichenen Spalte eine 1 hatten. Falls in y_i mindestens die Hälfte aller Einträge 1 war, wird mindestens die Hälfte aller Zeilen von M_i gestrichen und in M_{i+1} hat wieder jede Zeile die Eigenschaft, dass mindestens die Hälfte aller Einträge 1 ist.

Also hat man nach spätestens $\log_2 |A|^n = cn \in O(n)$ Iterationen alle Zeilen gestrichen.

Eine „normale“ deterministische Turingmaschine D, die mit $y = y_0 \cdots y_{cn}$ als Hilfestellung jedes Wort der Länge n in Polynomialzeit auf Zugehörigkeit zu L überprüft, kann wie folgt arbeiten: Zu gegebener Eingabe x bestimmt D zunächst $n = |x|$ und zerlegt y in Teilwörter y_i passender Länge. Anschließend simuliert D alle $T(x, y_i)$ bis einmal akzeptiert wurde und akzeptiert dann auch, oder lehnt x ab, falls kein $T(x, y_i)$ akzeptiert.

Die Länge von y ist polynomiell in n und mit T arbeitet auch D in Polynomialzeit. ■

Um die Fehlerwahrscheinlichkeit auf 2^{-k} zu senken, haben wir in vorangegangenen Kapiteln einen gegebenen randomisierten Algorithmus mehrfach (k mal) ausgeführt und jedes Mal unabhängig neue Zufallsbits verwendet, insgesamt also $r \cdot k$ Zufallsbits. Im folgenden werden wir eine Möglichkeit kennenlernen, bei **BPP**-Algorithmen die Fehlerwahrscheinlichkeit mit Hilfe von $O(r)$ Zufallsbits auf $O(1/k)$ zu reduzieren (sofern $k \leqslant 2^r$ ist).

12.43 ALGORITHMUS. (CHOR UND GOLDREICH (1989)) Es sei T eine **BPP**-Turingmaschine für eine formale Sprache L, die für Eingaben der Länge n eine (polynomielle) Zahl r von Zufallsbits benötigt. Es sei H eine stark 2-universelle Familie von Hashfunktionen $h : \{0,1\}^r \to \{0,1\}^r$, etwa die aus Beispiel 12.37. Dann genügen $2r$ Zufallsbits (a, b) um zufällig gleichverteilt ein $h \in H$ auszuwählen.

Der neue Algorithmus arbeitet dann so:

- Wähle zufällig ein $h_{a,b}$ aus H aus.
- Berechne für $i = 1, \ldots, k$ jeweils $y_i = h_{a,b}(i)$.
- Zähle, für wieviele i gilt: $y_i \in W_x$.

 © Th. Worsch 2000-2015

- Falls das mindestens $k/2$ mal der Fall ist, wird x akzeptiert,
- andernfalls wird x abgelehnt.

12.44 SATZ. *Die Fehlerwahrscheinlichkeit von Algorithmus 12.43 ist* $O(1/k)$.

12.45 BEWEIS. Man betrachte die Zufallsvariablen

$$X_i = \begin{cases} 1 & \text{falls } y_i \in W_x \\ 0 & \text{sonst} \end{cases} .$$

Sie sind identisch verteilt und paarweise unabhängig, denn

$$\begin{aligned}
\mathbf{Pr}\left[X_i = a \wedge X_j = b\right] &= \mathbf{Pr}\left[[y_i \in W_x] = a \wedge [y_j \in W_x] = b\right] \\
&= \mathbf{Pr}\left[h(i) \in W_x] = a \wedge [h(j) \in W_x] = b\right] \\
&= \sum_{z \in W_x} \sum_{z' \in W_x} \mathbf{Pr}\left[[h(i) \in \{z\}] = a \wedge [h(j) \in \{z'\}] = b\right] \\
&= \sum_{z \in W_x} \sum_{z' \in W_x} \mathbf{Pr}\left[h[(i) \in \{z\}] = a\right] \cdot \mathbf{Pr}\left[[h(j) \in \{z'\}] = b\right] \\
&= \sum_{z \in W_x} \mathbf{Pr}\left[[h(i) \in \{z\}] = a\right] \cdot \sum_{z' \in W_x} \mathbf{Pr}\left[[h(j) \in \{z'\}] = b\right] \\
&= \mathbf{Pr}\left[[h(i) \in W_x] = a\right] \cdot \mathbf{Pr}\left[[h(j) \in W_x] = b\right] \\
&= \mathbf{Pr}\left[X_i = a\right] \cdot \mathbf{Pr}\left[X_j = b\right]
\end{aligned}$$

Der Erwartungswert ist $\mu = \mathbf{E}[X_i] = \mu(W_x)$ und die Varianz $\mathbf{var}[X_i] = \mathbf{E}\left[X_i^2\right] - (\mathbf{E}[X_i])^2 = \mathbf{E}[X_i] - (\mathbf{E}[X_i])^2 = \mu(1-\mu)$. Da $0 \leqslant \mu \leqslant 1$, ist $\mathbf{var}[X_i] \leqslant 1/4$. Also ist $\mathbf{var}[\sum X_i] \leqslant k/4$.

Der Algorithmus liefert das falsche Ergebnis, falls

- $x \in L$ und $\sum X_i < k/2$ oder $x \notin L$ und $\sum X_i \geqslant k/2$, d. h.
- $\mu(W_x) > 3/4$ und $\sum X_i < k/2$ oder $\mu(W_x) < 1/4$ und $\sum X_i \geqslant k/2$, d. h.
- $k\mu(W_x) > 3k/4$ und $\sum X_i < k/2$ oder $k\mu(W_x) < k/4$ und $\sum X_i \geqslant k/2$.

Beide Fälle können zu der Form $|\sum X_i - \mathbf{E}[\sum X_i]| > k/4$ zusammen gefasst werden. Nach der Ungleichung von Chebyshev kann man die Fehlerwahrscheinlichkeit abschätzen durch

$$\mathbf{Pr}\left[\left|\sum X_i - \mathbf{E}\left[\sum X_i\right]\right| > k/4\right] \leqslant \frac{16\,\mathbf{var}[\sum X_i]}{k^2} \leqslant \frac{4}{k} .$$

■

Literatur

Adleman, Leonard (1978). „Two Theorems on Random Polynomial Time“. In: *19th Annual Symposium on Foundations of Computer Science (FOCS '78)*. IEEE Computer Society Press, S. 75–83 (siehe S. 114).

Chor, Benny und Oded Goldreich (1989). „On the Power of Two-Point Based Sampling“. In: *Journal of Complexity* 5.1, S. 96–106 (siehe S. 114).

Fredman, Michael L., János Komlós und Endre Szemerédi (1984). „Storing a Sparse Table with O(1) Worst Case Access Time“. In: *Journal of the ACM* 31.2, S. 538–544 (siehe S. 110).

 © Th. Worsch 2000-2015

Goldreich, Oded (2001). *Randomized Methods in Computation*. Lecture notes. URL: http://www.wisdom.weizmann.ac.il/~oded/rnd-sum.html (siehe S. 112).

Luby, Michael und Avi Wigderson (1995). *Pairwise Independence and Derandomization*. Technical report TR-95-035. International Computer Science Institute. URL: http://www.icsi.berkeley.edu/~luby/pair_sur.html (siehe S. 112).

 © Th. Worsch 2000-2015

13 Pseudo-Zufallszahlen

13.1 Die folgenden Worte werden John von Neumann zugeschrieben:

> *Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*

Trotzdem werden heute sehr oft Zahlen mit dem Rechner erzeugt, die „so aussehen, als ob" sie zufällig wären. Die Erfahrungen und Erkenntnisse der letzten Jahre und Jahrzehnte zeigen allerdings, dass es sehr viel schwerer ist, „gute" Pseudozufallszahlen[1] zu erzeugen als man vielleicht meint. Die folgenden Abschnitte sollen einen kleinen Einblick geben, damit man in der Praxis zumindest die gröbsten Anfängerfehler vermeiden kann.

13.2 Man findet das Zitat aus Punkt 13.1 zum Beispiel am Anfang von Kapitel 3 von TAOCP (Donald E. Knuth 1998), das sich mit Zufallszahlen beschäftigt.

Einen guten Überblick gibt auch die Arbeit von L'Ecuyer (1998), deren Text auch unter `http://www.iro.umontreal.ca/~lecuyer/myftp/papers/handsim.ps` erhältlich ist. Vom gleichen Autor stammt auch `http://www.iro.umontreal.ca/~lecuyer/myftp/papers/tutaor.ps` (L'Ecuyer 1994).

Unter `http://random.mat.sbg.ac.at/results/karl/server/` findet man unter anderem Informationen über eine ganze Reihe von in der Praxis verwendeten Zufallszahlengeneratoren.

13.3 Wegen der Probleme, die man sich mit (schlechten) Pseudozufallszahlen einhandeln kann, könnte man daran denken, statt dessen *echte* Zufallszahlen zu verwenden, die von einem physikalischen Prozess erzeugt werden.

Dieses Verfahren hat aber Nachteile. Wir kommen im Anschluss an die Aufzählung wünschenswerter Eigenschaften von Zufallszahlen noch einmal kurz darauf zu sprechen.

13.1 Allgemeines

13.4 An Pseudozufallszahlengeneratoren stellt man üblicherweise gewisse Anforderungen. Gewünschte Eigenschaften sind unter anderem:

- schnelle Erzeugbarkeit
- lange Periode
- Reproduzierbarkeit
- portable Implementierbarkeit
- schnelles Vorwärtsspringen
- „zufällige" Zahlen

Vom letzten Punkt abgesehen werden die Forderungen auch alle von der zyklischen Folge $0, 1, \ldots, m-1, 0, 1, 2, \ldots, m-1, \ldots$ erfüllt. Die große Aufgabe besteht daher darin festzulegen,

[1] Was auch immer das sein mag.

inwiefern die erzeugten Zahlen „zufällig“ sind; oder vielmehr „zufällig“ aussehen, denn erzeugen möchte man sie ja deterministisch.

Etwas konkreter möchte man von einfachen Generatoren verlangen, dass die Zufallszahlen zwei Eigenschaften haben:

- Die Zufallszahlen sollen (in einem gewissen Intervall, etwa $[0;1[$) gleichverteilt sein.
- Die Zufallszahlen sollen unabhängig voneinander sein.

Die erste Eigenschaft ist unter Umständen noch zu garantieren. Die zweite ist dagegen im allgemeinen verletzt (siehe Definition 13.6). Man ist zufrieden, wenn die erzeugten Zahlen „unabhängig aussehen“; genauer gesagt will man, dass man die statistischen Abhängigkeiten nicht mit einfachen (i. e. schnellen) Tests feststellen kann.

Wir werden in Abschnitt 13.3 auf diese schwierigen Aspekte zurückkommen. Insbesondere ist es so, dass diese oder weitere gewünschte Eigenschaften davon abhängen, in welchem Zusammenhang man die Pseudozufallszahlen verwenden will. Es gibt zum Beispiel Generatoren, die als sehr gut geeignet z. B. für Monte-Carlo-Simulationen gelten, aber für kryptographische Zwecke ungeeignet sind.

13.5 Man kann nun auch einige Nachteile nennen, die physikalische Prozesse zur Erzeugung von Zufallszahlen häufig haben:

- Sie sind nicht gleichverteilt.
- Sie sind nicht unabhängig.
- Sie sind nicht reproduzierbar.

13.6 Definition Ein (Pseudo-)Zufallszahlengenerator (RNG) ist eine Struktur $G = (S, s_0, T, U, G)$ mit folgenden Komponenten:

- einer endliche Menge S von *Zuständen*,
- einem *Anfangszustand* $s_0 \in S$,
- einer *Überführungsfunktion* $T : S \to S$,
- einer Menge U von *Ausgabewerten* und
- einer *Ausgabefunktion* $G : S \to U$.

Der Generator beginnt in Zustand s_0 und durchläuft nacheinander die Zustände $s_1, s_2, \ldots$, mit $s_{i+1} = T(s_i)$ für alle $i \geqslant 0$. In Zustand s_i wird die Ausgabe $u_i = G(s_i)$ produziert. ◇

13.7 Da die Zustandsmenge S eines Zufallszahlengenerators endlich ist, muss die Folge $s_0, s_1, s_2, \ldots$ und damit auch die Folge der Ausgaben zyklisch werden. Die Länge ρ des Zyklus heißt auch *Periodenlänge* des Generators.

13.8 Viele Generatoren sind von der folgenden Bauart: Zustandsmenge sind die nichtnegativen ganzen Zahlen kleiner einem $m \in \mathbb{N}$. Die Ausgabewerte sind die reellen Zahlen in dem Intervall $[0;1[$ und die Ausgabefunktion ist $u_i = s_i/m$.

Bei etwas aufwendigeren Generatoren besteht jeder Zustand aus $k > 1$ Zahlen $(x_i, x_{i-1}, \ldots, x_{i-k+1})$ kleiner m und die Ausgabefunktion ist z. B. $u_i = x_i/m$.

 © Th. Worsch 2000-2015

13.2 Generatoren für Pseudo-Zufallszahlen

13.2.1 Middle-square- und Muddle-square-Generator

13.9 Von Neumanns *Middle-square-Generator* sieht auf den ersten Blick zwar ganz nett aus, ist aber praktisch unbrauchbar.

Der Generator funktioniert so: Man arbeitet mit b-Bit Zahlen. Der Startwert ist x_0. Aus x_{n-1} ergibt sich x_n wie folgt: Man schreibt das Quadrat von x_{n-1} als $2b$-Bit Zahl. Daraus werden die mittleren b Bits extrahiert; sie bilden x_n.

Von der Verwendung dieses Generators wird dringend abgeraten, weil seine Periodenlänge zu klein ist.

13.10 Interessanterweise gibt es Modifikationen hiervon, die Donald E. Knuth (1998) als *Muddle-square-Generator* bezeichnet. Er stammt von Blum, Blum und Shub bzw. eine Verallgemeinerung von Leonid Levin. Letztere ist wie folgt definiert:

Alle Zahlen sind r Bits lang. Es sei m Produkt zweier großer Primzahlen der Form $4\ell + 3$. x_0 sei relativ prim zu m. Außerdem ist eine Zahl z als Maske gegeben. Sind die Dualzahldarstellungen $x = (a_{r-1} \cdots a_0)_2$ und $z = (b_{r-1} \cdots b_0)_2$ gegeben, so sei $x \cdot z = a_{r-1}b_{r-1} + \cdots + a_0 b_0$. Dann ist Levins Generator gegeben durch

$$\begin{aligned} x_{n+1} &= x_n^2 \pmod{m} \\ u_{n+1} &= x_{n+1} \cdot z \pmod{2} \end{aligned}$$

Man kann zeigen, dass der Generator bei zufälliger Wahl von x_0, m und z alle statistischen Test übersteht, die nicht mehr Zeit benötigen als das Faktorisieren natürlicher Zahlen.

Der Generator von Blum/Blum/Shub ist der Spezialfall von Levins Generator für $z = 1$.

In der Praxis scheinen diese Generatoren bislang keine Bedeutung zu spielen.

13.2.2 Lineare Kongruenzgeneratoren

13.11 Definition Ein *linearer Kongruenzgenerator* (engl. *linear congruence generator*, LCG) ist gegeben durch einen Anfangswert x_0, Modulus m, Multiplikator a und Inkrement c:

$$x_n = a x_{n-1} + c \pmod{m} \tag{13.1}$$

Ein LCG heißt *multiplikativ* oder ein MLCG, wenn $c = 0$ ist. ◇

13.12 Satz. *Der LCG aus Gleichung 13.1 hat genau dann volle Periodenlänge, wenn gilt:*

- $\operatorname{ggt}(m, c) = 1$,
- q *prim und* $q|m \implies q|a-1$ *und*
- $4|m \implies 4|a-1$.

13.13 Ein schlechter Generator war RANDU, der lange im IBM/360 Betriebssystem benutzt wurde. Bei ihm ist $a = 65539$, $c = 0$ und $m = 2^{31}$. Der Generator hat Periodenlänge 2^{29} und eine schlechte Gitterstruktur (siehe Abschnitt 13.3).

 © Th. Worsch 2000-2015

13.14 Der ANSI-C Generator `rand` ist der LCG mit $m = 2^{31}$, $a = 1\,103\,515\,245$, $c = 12345$ und $x_0 = 12345$. Dieser Generator ist schlecht. Die niedrigwertigen Bits der erzeugten Zahlen sind „nicht sehr zufällig“.

Der Generator `drand48` von Solaris ist ebenfalls ein LCG, und zwar der mit Parametern $a = 25\,214\,903\,917$, $c = 11$ und $m = 2^{48}$. Er hat volle Periodenlänge. Er ist besser als `rand`, aber auch er besteht manche statistische Tests nicht.

13.15 Ein besserer LCG ist der mit $m = 2^{32}$, $a = 69069$ und $c = 1$ von Marsaglia.

13.2.3 Mehrfach rekursive Generatoren

13.16 Definition Ein *mehrfach rekursiver Generator* (MRG) eine Verallgemeinerung eines MLCG. Hier ist

$$x_n = a_1 x_{n-1} + \cdots + a_k x_{n-k+1} \pmod m \tag{13.2}$$

wobei die Arithmetik modulo m durchgeführt werde und die Konstanten $a_1, \ldots, a_k$ aus dem Bereich $\{-(m-1), \ldots, m-1\}$ stammen. ◇

13.17 Definition Spezialfälle der MRG sind die sogenannten *lagged Fibonacci generators* (LFG). Der Name rührt von der einfachen Version

$$x_n = x_{n-1} + x_{n-2} \pmod m$$

her, die aber zu schlecht ist. Statt dessen empfiehlt sich

$$x_n = x_{n-r} + x_{n-s} \pmod m$$

für geeignete Zahlen r und s. Außerdem kann man statt der Addition auch andere Operationen in Betracht ziehen. ◇

13.18 Exklusives Oder in LFG hat sich als schlecht herausgestellt. Lange Zeit galt $(r, s) = (24, 55)$ und Addition als gute Wahl. Für $m = 2^e$ hat der Generator dann Periodenlänge $2^{e-1}(2^{55} - 1)$. Allerdings sind z. B. die niedrigstwertigen Bits der x_i nicht „gut“ verteilt. Marsaglia (1985) schlägt einige Varianten vor.

13.19 L'Ecuyer (1998) schlägt den folgenden MRG vor:

$$\begin{aligned} x_n \;=\; & 2\,620\,007\,610\,006\,878\,699\; x_{n-1} + \\ & 4\,374\,377\,652\,968\,432\,818\; x_{n-2} + \\ & \;\;667\,476\,516\,358\,487\,852\; x_{n-3} \qquad \text{mod } (2^{31}-1)(2^{31}-2000169) \end{aligned}$$

13.20 Satz. *Ein MRG hat maximale Periodenlänge $\rho = m^k - 1$, falls das sogenannte* charakteristische Polynom $P(z) = z^k - a_1 z^{k-1} - \cdots - a_k$ *primitives Polynom des Körpers $\mathbb{Z}_m$ ist.*

13.2.4 Inverse Kongruenzgeneratoren

13.21 Definition Ein *inverser Kongruenzgenerator* (ICG, Eichenauer und Lehn 1986) ist durch die Gleichung

$$x_{n+1} = (a x_n^{-1} + c) \pmod p$$

festgelegt, wobei p eine Primzahl sei. Da ein $x_i = 0$ sein kann, werde festgelegt: $0^{-1} = 0$. ◇

 © Th. Worsch 2000-2015

13.22 Diese Generatoren haben zum einen bemerkenswert gute theoretische Eigenschaften. Zum anderen bestehen aber auch empirische Tests mit sehr guten Ergebnissen. Ein kleiner Nachteil von ICG ist die nicht ganz so einfache Implementierung (Geschwindigkeit des Generators?).

Konkrete Parameter für gute ICG findet man im WWW unter http://random.mat.sbg.ac.at/generators/wsc95/inversive/.

13.2.5 Mersenne-Twister

13.23 Definition Der *Mersenne-Twister* stammt von Matsumoto und Nishimura (1998))(Artikel verfügbar unter http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf) und ist wie folgt definiert. Alle Zahlen sind Bitvektoren der Länge w und r ist eine Zahl mit $0 \leqslant r \leqslant w-1$. Es sind zwei Zahlen k und m mit $1 \leqslant m < k$ festgelegt und eine $w \times w$-Matrix A ist geeignet gewählt.

Es bezeichne x^l_{n+1} die „unteren" (lower) r Bits von x_{n+1} und x^u_n die „oberen" (upper) $w-r$ Bits von x_n. Außerdem bezeichne $x^u_n|x^l_{n+1}$ die Konkatenation der beiden Bitfolgen. Dann ist

$$x_{n+k} = x_{n+m} \oplus (x^u_n|x^l_{n+1})A \ .$$

◇

13.24 Eine konkrete Implementierung MT19937 eines Mersenne-Twisters hat Periodenlänge $2^{19937}-1$ und auch ausgezeichnete Eigenschaften hinsichtlich statistischer Tests.

Weitere Informationen einschließlich der Implementierungen in verschiedenen Programmiersprachen findet man über die Seite http://www.math.keio.ac.jp/~matumoto/emt.html im WWW.

13.25 Aber auch beim Mersenne-Twister gibt es noch Raum für Verbesserungen. Man lese etwa den Aufsatz von Panneton, L'Ecyuer und Matsumoto (2006). Dort werden Generatoren (WELL) vorgeschlagen, die zum Teil noch längere Periode, und dabei aber noch bessere Eigenschaften als der Mersenne-Twister haben. Implementierungen findet man z. B. unter http://www.iro.umontreal.ca/~panneton/WELLRNG.html.

13.2.6 Zellularautomaten als Generatoren

Manche Zellularautomaten zeigen, ausgehend selbst von „einfachen" Anfangskonfigurationen, „zufälliges" Verhalten. Zum Beispiel wird in Mathematica der folgende besonders einfache Generator benutzt:

13.26 Definition Gegeben ist ein Feld von k Bits $x_0, x_1, \dots, x_{k-1}$. Die sogenannte *Regel 30* besagt, dass in einem Schritt gleichzeitig neue Werte $y_0, \dots, y_{k-1}$ nach der folgenden Vorschrift berechnet werden:

x_{i-1}	1	0	1	0	1	0	1	0
x_i	1	1	0	0	1	1	0	0
x_{i+1}	1	1	1	1	0	0	0	0
y_i	0	0	0	1	1	1	1	0

Dabei sind die Indexrechnungen $i-1$ und $i+1$ modulo k zu verstehen. Die so berechneten y_i bilden die Werte x_i für den nächsten Schritt.

Der Name „Regel 30" rührt daher, dass die Folge 00011110 der Bits für y_i die Dualzahldarstellung der Zahl 30 ist. ◇

 © Th. Worsch 2000-2015

13.27 Wenn man k groß genug wählt (in Mathematica einige Hundert), in der Anfangskonfiguration mindestens eine 1 hat, zunächst eine hinreichende große Zahl Schritten zur Initialisierung durchläuft, und dann in einer willkürlich aber fest gewählten Zelle die Folge der durchlaufenen Werte benutzt, erhält man einen Generator, der z. B. alle von Marsaglia's `diehard`-Tests besteht (Schloissnig 2003).

13.2.7 Kombination mehrerer Generatoren

Es ist eine recht naheliegende Idee, durch geeignete „Kombination" zweier oder mehrerer Generatoren neue zu erhalten, die ein „besseres" Verhalten aufweisen, als ihre Komponenten.

Zum Beispiel ist in vielen Fällen leicht einzusehen, dass die Periodenlänge größer wird. Manchmal zeigen theoretische Überlegungen oder empirische Tests, dass sich auch das statistische Verhalten verbessert.

Zwei Kombinationsverfahren werden besonders gerne verwendet.

13.28 Beim sogenannten *Shuffling* werden zwei Generatoren für Zahlen x_i und y_i benutzt. Man unterhält eine Tabelle mit einer gewissen Anzahl zuletzt erzeugter x_i. Wann immer ein weiterer Wert benötigt wird, benutzt man das nächste y_j, um zufällig einen Wert aus der Tabelle auszuwählen. Der entsprechende Platz wird außerdem mit dem nächsten neuen x_i gefüllt.

Diese Methode hat zwei Nachteile. Zum einen ist sie theoretisch nicht gut verstanden, zum anderen ist es unbekannt, wie man schnell viele Zufallswerte überspringen kann.

13.29 Die andere Klasse von Methoden besteht darin, aus zwei Zufallsfolgen $x_0, x_1, \ldots$ und $y_0, y_1, \ldots$ mittels einer Operation $\bullet$ eine neue Folge $x_0 \bullet y_0, x_1 \bullet y_1, \ldots$ zu erzeugen. Die Operation kann Addition, bitweises Exklusives Oder, usw. sein.

Auch bei dieser Vorgehensweise vergrößert sich (bei „vernünftiger" Vorgehensweise) die Periodenlänge. Außerdem wird die Uniformität der Verteilung besser (oder jedenfalls nicht schlechter) und die Unabhängigkeit aufeinanderfolgender Werte größer (siehe z. B. Marsaglia 1985).

 © Th. Worsch 2000-2015

13.3 Tests für Pseudo-Zufallszahlen

Wie stellt man fest, ob ein Pseudozufallszahlengenerator „gut“ ist? Was heißt überhaupt „gut“?

Um das zu präzisieren, hat es sich eingebürgert, Generatoren verschiedenen Tests zu unterziehen.

13.30 Ergebnis solcher Tests sind üblicherweise Zahlen (etwa zwischen 0 und 1), die in naheliegender Weise als Qualitätsangabe interpretiert werden können. Manchmal ist es möglich, solche Zahlen auf Grund der Definition des Generators mathematisch herzuleiten. Man spricht dann von einem *theoretischen Test*. In anderen Fällen muss man *empirische Test* durchführen, das heißt den Generator zur Erzeugung vieler Pseudozufallszahlen verwenden und die konkreten Ergebnisse untersuchen.

13.3.1 Grundsätzliches: χ^2- und KS-Test

13.31 Definition Es seien $s_1, \dots, s_k$ die möglichen Ergebnisse eines Zufallsexperiments, die mit Wahrscheinlichkeiten $p_1, \dots, p_k$ auftreten. Für eine Reihe von n *unabhängigen* Zufallsexperimenten bezeichne Y_i die absolute Häufigkeit, mit der Ergebnis s_i auftrat. Es ist also $\sum_i Y_i = n$ und $\mathbf{E}[Y_i] = np_i$.

Dann bezeichnet man

$$V = \sum_{i=1}^{k} \frac{(Y_i - \mathbf{E}[Y_i])^2}{\mathbf{E}[Y_i]}$$

als χ^2-Statistik der beobachteten Größen $Y_1, \dots, Y_k$. ◇

13.32 Setzt man in die obige Formel $\mathbf{E}[Y_i] = np_i$ ein, so erhält man

$$V = \sum_{i=1}^{k} \frac{(Y_i - np_i)^2}{np_i} = \sum_{i=1}^{k} \frac{Y_i^2 - 2Y_i np_i + n^2 p_i^2}{np_i} = \frac{1}{n} \sum_{i=1}^{k} \frac{Y_i^2}{p_i} - 2n + n = \frac{1}{n} \sum_{i=1}^{k} \frac{Y_i^2}{p_i} - n\,.$$

13.33 Es stellt sich nun die Frage, wie die Größe V verteilt ist. Hierfür gibt es Tabellen mit Näherungen. Sie enthalten für jeden Wert $\nu = k - 1$, die sogenannte *Anzahl der Freiheitsgrade*, eine Zeile und für einige Wahrscheinlichkeitswerte p, z. B. $p = 1\%$, $p = 5\%$, ..., $p = 99\%$ eine Spalte. Man beachte, dass die Anzahl n nicht eingeht; sie muss groß genug gewählt sein. Eine Daumenregel besagt, dass n so groß sein muss, dass jedes $s_i \geqslant 5$ ist.

In einem solchen Fall besagt ein Wert x in Zeile $\nu = k - 1$ und Spalte p: Der Wert V ist kleiner oder gleich x mit Wahrscheinlichkeit p (falls n groß genug ist).

13.34 Beispiel. Zum Beispiel findet man in einer Tabelle für χ^2:

	$p = 0.01$	$p = 0.05$	$p = 0.25$	$p = 0.5$	$p = 0.75$	$p = 0.95$	$p = 0.99$
$\nu = 11$	3.053	4.575	7.584	10.34	13.70	19.68	24.72

Ist etwa $k = 12$ und hat man etwa für eine Reihe von Versuchen einen Wert $V = 29.44$ errechnet, dann liest man aus der Tabelle ab: In 99% der Fälle ist $V \leqslant 24.72$, also ist $V > 24.72$ und erst recht $V > 29.44$ in höchsten einem Prozent der Fälle. Man wird es also als sehr unwahrscheinlich ansehen, dass die Versuchsreihe der angenommenen Verteilung genügt.

 © Th. Worsch 2000-2015

Andererseits betrachte man den Fall, dass die Y_i alle sehr sehr gut den Erwartungswerten entsprechen. Dann ist also V klein, etwa $V = 1.17$. Der Tabelle entnimmt man, dass auch das in weniger als einem Prozent der Fälle passiert.

Hat man es nicht mit diskreten, sondern mit kontinuierlich verteilten Werten zu tun, dann hilft der KS-Test.

13.35 Definition Der *Kolmogorov-Smirnov-Test* (kurz KS-Test) wird wie folgt durchgeführt. Gegeben sind n *unabhängige* Zufallsexperimente mit Ergebnissen $X_1, \dots, X_n$. Diese induzieren eine empirische Verteilungsfunktion

$$F_n(x) = \frac{|\{i \mid X_i \leqslant x\}|}{n} .$$

Zum Vergleich mit einer vorgegebenen kontinuierlichen Verteilungsfunktion $F(x)$ berechnet man die Größen

$$\begin{aligned} K_n^+ &= \sqrt{n} \max_{-\infty<x<+\infty} (F_n(x) - F(x)) \qquad \text{und} \\ K_n^- &= \sqrt{n} \max_{-\infty<x<+\infty} (F(x) - F_n(x)) \end{aligned}$$

Für diese Größen kann man wieder in einer Tabelle mit Zeilen für verschiedene n und Spalten für verschiedene p nachschlagen, mit welcher Wahrscheinlichkeit ein bestimmter K-Wert kleiner oder gleich (bzw. größer) einem bestimmten Wert ist. ◇

13.36 Eine mögliche Anwendung des KS-Test ist die folgende. Sinnvollerweise macht einen χ^2-Test nicht nur einmal, sondern mehrmals. Da die Verteilung für χ^2 (jedenfalls näherungsweise) bekannt sind, kann man auf die sich ergebenden Wahrscheinlichkeiten einen KS-Test anwenden.

Analog kennt man (näherungsweise für große n) die Verteilung der K_n und kann auf diese Werte einen weiteren KS-Test anwenden.

13.3.2 Einfache empirische Tests

Im folgenden gehen wir auf einige empirische Test für Zufallszahlengeneratoren ein. Zu den Zielen solcher Tests gehört es, Informationen über die vermutliche Art der Verteilung zu bekommen, die Un-/Abhängigkeit der erzeugten Zahlen, oder etwa eventuelle Korrelationen.

Es gibt frei verfügbare Programmpakete, die Zufallszahlengeneratoren einer Reihe von Tests unterziehen. Zum Beispiel ist unter `http://stat.fsu.edu/pub/diehard/` die Sammlung *Diehard* von George Marsaglia zu finden. Dabei handelt es sich „schwer" zu bestehende Tests (siehe Marsaglia 1985). Weitere Sammlungen finden sich bei Richard Simard unter `http://www.iro.umontreal.ca/~simardr/testu01/tu01.html` und beim NIST unter `http://csrc.nist.gov/rng/rng2.html`. TestU01 enthält auch die Implementierungen von fast zweihundert Pseudozufallszahlengeneratoren, von denen L'Ecuyer und Simard (2007) sagen: „some are good and many are bad".

13.37 Im folgenden gehen wir davon aus, dass eine Folge $U_0, U_1, \dots$ von reellen Zahlen vorliegt, die angeblich unabhängig und gleichverteilt sind zwischen 0 und 1.

Werden für einen Test nichtnegative ganze Zahlen benötigt, so werde statt dessen die Folge der $Y_i = \lfloor dU_i \rfloor$ für eine geeignete Konstante $d \in \mathbb{N}$ betrachtet.

13.38 Der einfachste Test besteht darin, zu versuchen festzustellen, ob unter den Y_i jeder der möglichen Werte $0, \dots, d-1$ gleich oft vorkommt.

 © Th. Worsch 2000-2015

13.39 Die analoge Eigenschaft kann man für Paare, Tripel, etc. prüfen. Man beachte, dass man dann die s-dimensionalen *nicht überlappenden* Vektoren $\mathbf{Y}_i = (Y_{si}, Y_{si+1}, \dots Y_{si+s-1})$ benutzen muss und *nicht* die überlappenden Vektoren $(Y_i, Y_{i+1}, \dots Y_{i+s-1})$.

Donald E. Knuth 1998 nennt dies den *Serial Test*.

13.40 Eine Variante dieses Tests für reelle Pseudozufallszahlen (L'Ecuyer 1998) besteht darin, den s-dimensionalen Einheitswürfel in $k = 2^{\ell s}$ gleich große Teilwürfel aufzuteilen und bei n zufällig gewählten Punkten für jeden Würfel j die Anzahl X_j der in ihm liegenden Punkte zu bestimmen. Für die Größe

$$\sum_{j=1}^{k} \frac{(X_j - n/k)^2}{n/k}$$

weiß man, dass ihr Erwartungswert $k-1$ und ihre Varianz $2(k-1)(n-1)/n$ sind, wenn die Punkte wirklich zufällig gleichverteilt liegen. Gegen diese Hypothese kann man die empirischen Daten testen.

13.41 Für den *Lückentest* (engl. *gap test*) sind zwei relle Zahlen $0 \leqslant \alpha < \beta \leqslant 1$ gegeben und man betrachtet die Längen der maximalen Teilfolgen $U_j, \dots, U_{j+r}$, so dass $\alpha \leqslant U_{j+r} \leqslant \beta$ aber die vorherigen Werte nicht. Man spricht dann von einer Lücke der Länge r.

Für $p = \beta - \alpha$ ist $p_r = p(1-p)^r$ die Wahrscheinlichkeit für eine Lücke der Länge r. Man kann daher leicht den χ^2-Test anwenden.

13.42 Beim *Permutationstest* betrachtet man nicht überlappende Vektoren $\mathbf{U}_i = (U_{si}, U_{si+1}, \dots U_{si+s-1})$ der Länge s. Für jeden Vektor wird die Permutation π der Indizes bestimmt, die die Komponenten sortiert. Es wird eine Statistik erstellt, welche Permutation wie häufig auftritt, und getestet, inwieweit das mit der theoretisch gegebenen Gleichverteilung verträglich ist.

13.43 Beim *Maximum-von-t Test* werden die Größen $V_i = \max\{U_{si}, U_{si+1}, \dots U_{si+s-1}\}$ berechnet. Falls die Pseudozufallszahlen wirklich gleichverteilt sind, ist

$$\mathbf{Pr}\left[V_i \leqslant x\right] = \mathbf{Pr}\left[\max\{U_{si}, U_{si+1}, \dots U_{si+s-1}\} \leqslant x\right] = \prod \mathbf{Pr}\left[U_{si+j} \leqslant x\right] = x^s .$$

Diese Hypothese kann mit einem KS-Test überprüft werden.

13.44 Der *Geburtstagsabstandstest* (engl. *birthday spacings test*) von Marsaglia arbeitet wie folgt: Gegebene ganze Pseudozufallszahlen $Y_1, Y_2, \dots, Y_m$ aus dem Bereich $\{1, \dots, n\}$ werden zunächst sortiert. Es bezeichne $Z_1, Z_2, \dots, Z_m$ die aufsteigend sortierte Reihenfolge der gleichen Zahlen. Hieraus werden die Abstände $S_i = Z_{i+1} - Z_i$ berechnet. Es bezeichne $R = \left|\{S_i \mid \exists j < i : S_j = S_i\}\right|$ die Anzahl der mehrfach vorkommenden Abstände. Man kann zeigen, dass R näherungsweise Poisson-verteilt ist mit Parameter $\lambda = m^3/(4n)$.

Man kann nun (bei geeignetem d) für die diskreten Ereignisse $R = 0$, $R = 1, \dots, R = d-1$ und $R \geqslant d$ die Wahrscheinlichkeiten ausrechnen, hinreichend oft in einem Experiment einen konkreten Wert R bestimmen und für die Ergebnisse einen χ^2-Test machen.

Lagged-Fibonacci-Generatoren haben üblicherweise Probleme mit dem Geburtstagsabstandstest, auch wenn sie viele andere Test bestehen (das gilt z. B. auch für den schon in Punkt 13.18 erwähnten Generator $x_n = x_{n-24} + x_{n-55} \pmod{2^e}$).

 © Th. Worsch 2000-2015

13.3.3 Weitere Tests

13.45 Für die Erklärung des Spektraltests betrachten wir zunächst den (zugegebenermaßen besonders) schlechten Pseudozufallszahlengenerator, der durch $x_{n+1} = 85x_n + 2 \pmod{256}$ und die Ausgabefunktion $u_n = x_n/256$ gegeben ist. Für mehr als 256 überlappende Paare (u_i, u_{i+1}) ihn sind in Abbildung 13.1 die entsprechenden Punkte im Einheitsquadrat eingezeichnet. Wie man sieht,

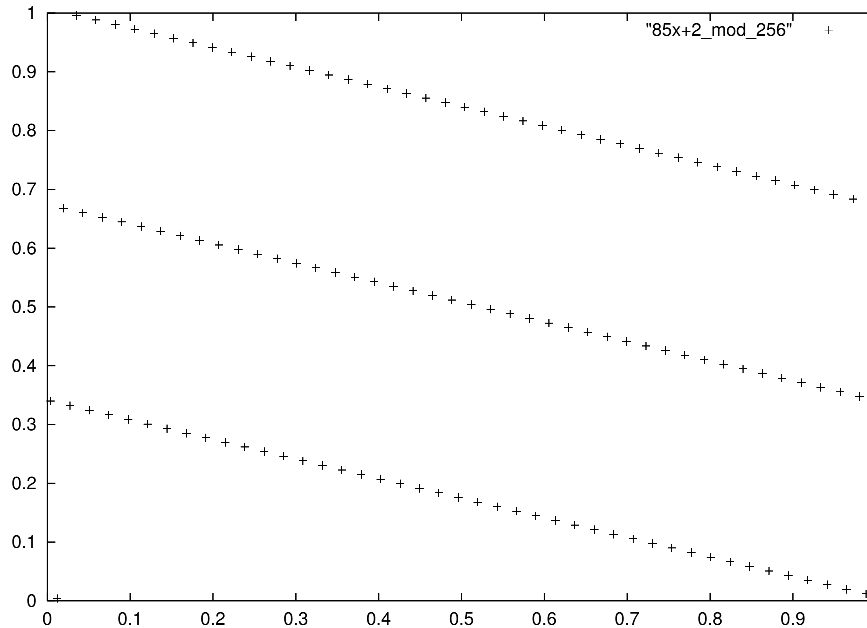


Abbildung 13.1: Gitterstruktur bei einem sehr schlechten Pseudozufallszahlengenerator.

sind die Punkte überhaupt nicht gleichmäßig und schon gar nicht „zufällig" verteilt.

Die zu beobachtende *Gitterstruktur* tritt bei linearen Kongruenzgeneratoren aber auch bei einigen anderen Generatorarten auf.

13.46 Der *Spektraltest* für Dimension s besteht darin, für s-dimensionale überlappende Vektoren $\mathbf{u}_i = (u_i, u_{i+1}, \dots, u_{i+s-1})$ im Einheitshyperwürfel die Zahl d_s zu ermitteln. Dabei ist d_s der maximale Abstand zwischen zwei Hyperebenen, genommen über alle Familien paralleler Hyperebenen im Einheitswürfel, die alle Punkte $\mathbf{u}_i$ beinhalten.

Je kleiner ein d_s ist, als desto „besser" wird man den verwendeten Generator (hinsichtlich Dimension s) ansehen.

13.47 Ein anderes „Qualitätsmaß" für Pseudozufallszahlengeneratoren ist die *Diskrepanz* (und als Sonderfall die *Sterndiskrepanz* (engl. *star discrepancy*)) (siehe auch Niederreiter 1992).

Hierfür betrachtet man wieder die Vektoren $\mathbf{u}_i = (u_i, u_{i+1}, \dots, u_{i+s-1})$ im Einheitshyperwürfel. Es seien N solche Punkte gegeben. Für jede Menge $R = \prod_{j=1}^{s} [\alpha_j, \beta_j[$ mit $0 \leqslant \alpha_j < \beta_j \leqslant 1$ sei $I(R)$ die Anzahl der $\mathbf{u}_i$, die in R liegen, und $V(R) = \prod_{j=1}^{s} (\beta_j - \alpha_j)$ das Volumen von R.

Die s-dimensionale Diskrepanz $D_N^{(s)}$ der Punkte $\mathbf{u}_0, \dots, \mathbf{u}_{N-1}$ ist

$$D_N^{(s)} = \max_R |V(R) - I(R)/N| \ .$$

 © Th. Worsch 2000-2015

Falls man sich auf Mengen R einschränkt, deren eine Ecke der Nullpunkt ist, falls also $\alpha_j = 0$ für alle j ist, dann spricht man von der Sterndiskrepanz $D_N^{*(s)}$.

Für Punktemengen, deren Verteilung „weit" von Gleichverteilung entfernt ist, ergeben sich zu große Werte bei der Diskrepanz, und wenn die Verteilung „zu gleichmäßig" ist, zu kleine Werte.

13.48 Beim *Näheste Paare Test* (engl. *nearest pair test*) erzeugt man zufällig n Punkte im s-dimensionalen Einheitshyperwürfel und bestimmt das Minimum D der (euklidischen) Abstände zwischen je zwei Punkten. Man kann zeigen, dass für große n die Zufallsvariable $T = n^2 D^s/2$ exponentiell verteilt ist mit Erwartungswert $1/V_s$, wobei V_s das Volumen der s-dimensionalen Einheitskugel ist.

Für einen Test erzeugt man N Werte für T und vergleicht sie mit der Exponentialverteilung.

13.49 Die *Ränge zufälliger Boolescher Matrizen* können ebenfalls zum Test herangezogen werden. Eine echt zufällig mit Nullen und Einsen gefüllte $m \times n$-Matrix hat Rang r mit $1 \leqslant r \leqslant \min(m, n)$ mit Wahrscheinlichkeit

$$2^{-(n-r)(m-r)} \prod_{i=0}^{r-1} \frac{(1-2^{i-n})(1-2^{i-m})}{1-2^{i-r}} .$$

Mit diesen Werten können empirisch ermittelte Zahlen verglichen werden. Insbesondere ist es interessant, für die Bits jeweils einer Zeile der Matrix die Bits *einer* Pseudozufallszahl zu benutzen.

13.50 Beim *OPSO-Test* (engl. *overlapping pairs sparse occupancy test*) werden jeweils 10 Bits (z. B., aus einer Pseudozufallszahl) als ein Symbol aus einem 1024-elementigen Alphabet angesehen. Es werden $2^{21} + 1$ solche Symbole erzeugt und für alle 2^{21} überlappenden Paare darauf hin überprüft, welche konkreten Paare von Symbolen fehlen. Die Theorie besagt, dass deren Zahl normalverteilt ist mit Erwartungswert 141 909 und Standardabweichung 290. Damit können die empirischen Zahlen verglichen werden.

Marsaglia (1985) schlägt auch noch andere Tests vor, bei denen allerdings nicht klar ist, welches die korrekten theoretischen Werte sind, mit denen die empirischen Ergebnisse verglichen werden müssen. Die Vergleichswerte bestimmt er daher statt auf theoretischem Wege ebenfalls mit der Hilfe von Pseudozufallszahlengeneratoren. An dieser Stelle ist natürlich eine gewisse Skepsis angebracht.

Allerdings vertritt z. B. L'Ecuyer (1992) die Ansicht, dass dies zumindest akzeptabel sei, solange viele vermutlich gute Pseudozufallszahlengeneratoren zu übereinstimmenden Zahlen kämen, die als Ersatz für die fehlenden theoretischen Werte genommen würden.

13.51 Als Beispiel für einen solchen Test sei hier der *parking lot test* genannt. Dabei ist z. B. ein Quadrat mit Seitenlänge 100 vorgegeben. Auf dieser Fläche sollen nun Einheitskreise („Hubschrauber von oben gesehen") an zufällig gewählten Stellen $\mathbf{u}_i$ positioniert („geparkt") werden. Damit ist gemeint, dass $\mathbf{u}_i$ als Mittelpunkt des Kreises gewählt wird, sofern er sich nicht mit anderen, bereits positionierten Kreisen schneidet. Ansonsten liegt eine Kollision vor und es wird nicht geparkt.

Man macht eine gewisse Anzahl n von Versuchen und zählt, wie oft ein Kreis ohne Kollision positioniert werden konnte.

13.52 Das gleiche Problem hat der *OQSO Test* (engl. *overlapping quadruples sparse occupancy test*). Er ist analog zum OPSO-Test benutzt aber Quadrupel von Symbolen über einem 32-elementigem Alphabet.

 © Th. Worsch 2000-2015

Literatur

Eichenauer, J. und J. Lehn (1986). „A non-linear congruential pseudo random number generator". In: *Statistische Hefte* 27, S. 315–326 (siehe S. 120).

Knuth, Donald E. (1998). *The Art of Computer Programming*. 3rd. Bd. 2. Addison-Wesley (siehe S. 117, 119, 125).

L'Ecuyer, Pierre (1992). „Testing Random Number Generators". In: *Winter Simulation Conference*. ACM, S. 305–313. ISBN: 0-7803-0798-4 (siehe S. 127).

— (1994). „Uniform Random Number Generation". In: *Annals of Operations Research* 53, S. 77–120 (siehe S. 117).

— (1998). „Random Number Generation". In: *Handbook of Simulation*. Wiley. Kap. 4, S. 93–137 (siehe S. 117, 120, 125).

L'Ecuyer, Pierre und Richard Simard (2007). „TestU01: A C Library for Empirical Testing of Random Number Generators". In: *ACM Transactions on Mathematical Software* 33.4 (siehe S. 124).

Marsaglia, George (1985). „A Current View of Random Number Generation". In: *Computer Science and Statistics, 16th Symposium on the Interface*. Elsevier Science Publishers, S. 3–10 (siehe S. 120, 122, 124, 127).

Matsumoto, Makoto und Takuji Nishimura (1998). „Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Transactions on Modeling and Computer Simulation* 8.1, S. 3–30 (siehe S. 121).

Niederreiter, Harald (1992). *Random Number Generation and Quasi-Monte Carlo Methods*. Bd. 63. CBMS-NSF regional conference series in Appl. Math. SIAM (siehe S. 126).

Panneton, Francois, Pierre L'Ecyuer und Makoto Matsumoto (2006). „Improved Long-Period Generators Based on Linear Recurrences Modulo 2". In: *ACM Transactions on Mathematical Software* 32.1, S. 1–16. URL: %5Curl%7Bhttp://www.iro.umontreal.ca/~lecuyer/myftp/papers/wellrng.pdf%7D (siehe S. 121).

Schloissnig, Siegfried (2003). „Zellularautomaten und Zufallszahlen". Studienarbeit. Univ. Karlsruhe, Fak. f. Informatik (siehe S. 122).

 © Th. Worsch 2000-2015

A Wahrscheinlichkeitstheoretische Grundlagen

In diesem Kapitel sind zur Erinnerung einige Definitionen und Ergebnisse (ohne Beweise) zusammengestellt, die man z. B. in einer Vorlesung über Wahrscheinlichkeitstheorie kennengelernt haben sollte.

A.1 Allgemeines

A.1 Definition Eine *σ-Algebra* $(\Omega, \mathbb{E})$ über einem *Ergebnisraum* Ω ist eine Menge $\mathbb{E} \subseteq 2^\Omega$ von *Ereignissen* $E \in \mathbb{E}$ mit den Eigenschaften:

1. $\emptyset \in \mathbb{E}$
2. $E \in \mathbb{E} \implies \bar{E} \in \mathbb{E}$
3. $(\forall i \in \mathbb{N} : E_i \in \mathbb{E}) \implies \bigcup_{i \in \mathbb{N}} E_i \in \mathbb{E}$

Ein *Wahrscheinlichkeitsmaß* $\mathbf{Pr}[\cdot]$ auf einer σ-Algebra ist eine Abbildung $\mathbf{Pr} : \mathbb{E} \to \mathbb{R}_+$ mit den Eigenschaften:

1. $\forall E \in \mathbb{E} : 0 \leqslant \mathbf{Pr}[E] \leqslant 1$
2. $\mathbf{Pr}[\Omega] = 1$
3. für alle paarweise disjunkten Ereignisse E_i gilt: $\mathbf{Pr}[\bigcup_i E_i] = \sum_i \mathbf{Pr}[E_i]$.

Ein *Wahrscheinlichkeitsraum* $(\Omega, \mathbb{E}, \mathbf{Pr}[\cdot])$ ist eine σ-Algebra $(\Omega, \mathbb{E})$ mit einem darauf definierten Wahrscheinlichkeitsmaß $\mathbf{Pr}[\cdot]$. Ein Wahrscheinlichkeitsraum heißt *diskret*, falls Ω höchstens abzählbar ist und für alle $\omega \in \Omega$ gilt: $\{\omega\} \in \mathbb{E}$. ◇

A.2 In einem diskreten Wahrscheinlichkeitsraum ist stets $\mathbb{E} = 2^\Omega$.

A.3 Beispiel. Für die Vorlesung sind Wahrscheinlichkeitsräume der folgenden Art sehr wichtig: Es sei R ein randomisierter Algorithmus und x eine Eingabe für R. Dann gibt es im allgemeinen mehrere verschiedene konkret mögliche Berechnungen von R für x. Das können wie zum Beispiel beim randomisierten Quicksort (siehe Kapitel 2) endlich viele sein.

Es können aber selbst für eine einzelne Eingabe auch abzählbar unendlich viele. Als einfaches Beispiel denke man an den (zugegebenermaßen reichlich langweiligen) randomisierten Algorithmus, der für jede natürliche Zahl x als Eingabe so lange Zufallsbits „würfelt", bis die x zuletzt produzierten Bits alle gleich 1 waren, und als Ausgabe z. B. die Gesamtzahl der benötigten Bits liefert.

A.4 Übung. Man gebe für die beiden eben genannten Beispiele Wahrscheinlichkeitsräume an, die jeweils allen möglichen Berechnungen für eine Eingabe x entsprechen.

A.5 Lemma. (Einschluss-Ausschluss-Prinzip) Sind $E_1, \dots, E_k$ beliebige Ereignisse, dann gilt

$$\mathbf{Pr}[E_1 \cup E_2] = \mathbf{Pr}[E_1] + \mathbf{Pr}[E_2] - \mathbf{Pr}[E_1 \cap E_2]$$

und allgemeiner

$$\begin{aligned}\mathbf{Pr}\left[\bigcup_i E_i\right] &= \sum_i E_i - \sum_{i<j} \mathbf{Pr}\left[E_i \cap E_j\right] + \sum_{i<j<k} \mathbf{Pr}\left[E_i \cap E_j \cap E_k\right] - \cdots \\ &\quad +(-1)^{l+1} \sum_{i_1<i_2<\cdots<i_l} \mathbf{Pr}\left[\bigcap_{r=1}^{l} E_{i_r}\right] + \cdots\end{aligned}$$

A.6 Definition Die *bedingte Wahrscheinlichkeit* von E_1 unter der Bedingung E_2 mit $\mathbf{Pr}[E_2] > 0$ ist $\mathbf{Pr}[E_1 \mid E_2] := \mathbf{Pr}[E_1 \cap E_2] / \mathbf{Pr}[E_2]$. Ist $\mathbf{Pr}[E_2] = 0$, so sei $\mathbf{Pr}[E_1 \mid E_2] := 0$. ◇

A.7 Satz. *Ist $E_1, \dots, E_k$ eine Partitionierung von Ω und ist $\mathbf{Pr}[E] > 0$, dann gilt:*

$$\mathbf{Pr}[E] = \sum_{i=1}^{k} \mathbf{Pr}[E \mid E_i] \cdot \mathbf{Pr}[E_i]$$

A.8 Korollar. (Formel von Bayes) Ist $E_1, \dots, E_k$ eine Partitionierung von Ω und ist $\mathbf{Pr}[E] > 0$, dann gilt:

$$\mathbf{Pr}[E_i \mid E] = \frac{\mathbf{Pr}[E_i \cap E]}{\mathbf{Pr}[E]} = \frac{\mathbf{Pr}[E \mid E_i]\, \mathbf{Pr}[E_i]}{\sum_{j=1}^{k} \mathbf{Pr}\left[E \mid E_j\right] \mathbf{Pr}\left[E_j\right]} .$$

A.9 Definition Zwei Ereignisse E_1 und E_2 heißen *(stochastisch) unabhängig*, falls gilt: $\mathbf{Pr}[E_1 \cap E_2] = \mathbf{Pr}[E_1] \cdot \mathbf{Pr}[E_2]$.

Allgemeiner heißt eine Menge $\{E_i \mid i \in I\}$ *unabhängig*, falls für alle $S \subseteq I$ gilt:

$$\mathbf{Pr}\left[\bigcap_{i \in S} E_i\right] = \prod_{i \in S} \mathbf{Pr}[E_i] .$$

Die Ereignisse heißen k-*unabhängig*, wenn obige Gleichung für alle S einer Größe kleiner gleich k gilt. ◇

A.2 Zufallsvariablen

A.10 Definition Eine *Zufallsvariable* X ist eine Abbildung $X : \Omega \to \mathbb{R}$, so dass für alle Borelmengen $B \subseteq \mathbb{R}$ gilt: $\{\omega \in \Omega \mid X(\omega) \in B\} \in \mathbb{E}$.

Wir schreiben statt $\mathbf{Pr}[\{\omega \in \Omega \mid X(\omega) \leqslant x\}]$ kurz $\mathbf{Pr}[X \leqslant x]$ und analog $\mathbf{Pr}[X = x]$. Außerdem ist z. B. $\mathbf{Pr}[X \leqslant x \wedge Y \leqslant y]$ zu verstehen als $\mathbf{Pr}[\{\omega \in \Omega \mid X(\omega) \leqslant x\} \cap \{\omega \in \Omega \mid Y(\omega) \leqslant y\}]$. ◇

A.11 Wir gehen im folgenden stillschweigend davon aus, dass $\mathbf{Pr}[X \leqslant x]$ und $\mathbf{Pr}[X = x]$ stets existieren, sofern es nicht ohnehin klar ist, etwa wenn der Wahrscheinlichkeitsraum $(\Omega, \mathbb{E})$ diskret ist.

A.12 Beispiel. *Das* Beispiel für Zufallsvariablen in dieser Vorlesung schlechthin ist der Zeitbedarf eines randomisierten Algorithmus für eine konkrete Eingabe.

 © Th. Worsch 2000-2015

A.13 Übung. Man präzisiere die eben getroffene Aussage für die Wahrscheinlichkeitsräume aus Beispiel A.3.

Im Fall des randomisierten Quicksort mache man sich klar, dass der Erwartungswert für die Laufzeit nur von der Anzahl der Datenelemente, aber nicht von ihrer ursprünglichen Reihenfolge abhängt. Für den „Bit-Würfel-Algorithmus" versuche man, den Erwartungswert für die Laufzeit in Abhängigkeit von der Anzahl der zu produzierenden 1-Bits zu bestimmen.

A.14 Definition Eine Zufallsvariable ist *diskret*, falls ihr Wertebereich endlich oder abzählbar unendlich ist.

Die *Indikatorvariable* für ein Ereignis E ist die diskrete Zufallsvariable X mit

$$X(\omega) = \begin{cases} 1 & \text{falls } \omega \in E \\ 0 & \text{falls } \omega \notin E \end{cases}$$

◇

A.15 Definition Die *Verteilungsfunktion* F_X einer Zufallsvariablen X ist die Abbildung

$$F_X : \mathbb{R} \to [0,1] : x \mapsto \mathbf{Pr}[X \leqslant x] \ .$$

Die *Dichtefunktion* p_X einer Zufallsvariablen X ist die Abbildung

$$p_X : \mathbb{R} \to [0,1] : x \mapsto \mathbf{Pr}[X = x] \ .$$

◇

A.16 Definition Die *gemeinsame Verteilungsfunktion* $F_{X,Y}$ zweier Zufallsvariablen X und Y, die auf dem gleichen Ergebnisraum definiert sind, ist die Abbildung

$$F_{X,Y} : \mathbb{R} \times \mathbb{R} \to [0,1] : (x,y) \mapsto \mathbf{Pr}[X \leqslant x \wedge Y \leqslant y] \ .$$

Die *gemeinsame Dichtefunktion* $p_{X,Y}$ von X und Y ist die Abbildung

$$p_{X,Y} : \mathbb{R} \times \mathbb{R} \to [0,1] : (x,y) \mapsto \mathbf{Pr}[X = x \wedge Y = y] \ .$$

◇

A.17 Definition Zwei Zufallsvariablen X und Y heißen *unabhängig*, wenn für alle $x, y \in \mathbb{R}$ gilt:

$$\mathbf{Pr}[X = x \wedge Y = y] = \mathbf{Pr}[X = x] \cdot \mathbf{Pr}[Y = y] \ .$$

Allgemeiner heißt eine Menge $\{X_i \mid i \in I\}$ von Zufallsvariablen *unabhängig*, falls für alle $S \subseteq I$ und alle Mengen $\{x_i \in \mathbb{R} \mid i \in I\}$ gilt:

$$\mathbf{Pr}\left[\bigwedge_{i \in S} X_i = x_i\right] = \prod_{i \in S} \mathbf{Pr}[X_i = x_i] \ .$$

Die Zufallsvariablen heißen k-*unabhängig*, wenn obige Gleichung für alle S einer Größe kleiner gleich k gilt. ◇

A.18 Lemma. Zwei Zufallsvariablen X und Y sind genau dann unabhängig, wenn für alle $x, y \in \mathbb{R}$ gilt:

$$\mathbf{Pr}[X = x \mid Y = y] = \mathbf{Pr}[X = x] \ .$$

A.19 Definition Der *Erwartungswert* $\mathbf{E}[X]$ einer Zufallsvariablen X ist $\mathbf{E}[X] := \sum_{x \in \mathbb{R}} x \cdot p_X(x)$, sofern diese Summe absolut konvergiert. ◇

 © Th. Worsch 2000-2015

Absolute Konvergenz bedeutet, dass sogar $\sum_{x \in \mathbb{R}} |x| \cdot p_X(x)$ konvergiert. In diesem Fall ist $\mathbf{E}[X]$ tatsächlich unabhängig von der Reihenfolge der Summanden in $\sum_{x \in \mathbb{R}} x \cdot p_X(x)$.

A.20 Lemma. Für beliebige Zufallsvariablen $X_1, \ldots, X_k$ und beliebige lineare Funktionen h gilt:

$$\mathbf{E}[h(X_1, \ldots, X_k)] = h(\mathbf{E}[X_1], \ldots, \mathbf{E}[X_k]) .$$

A.21 Lemma. Für unabhängige Zufallsvariablen X und Y gilt:

$$\mathbf{E}[XY] = \mathbf{E}[X] \cdot \mathbf{E}[Y] .$$

A.22 Definition Für $k \in \mathbb{N}$ sind das k*te Moment* m_X^k und das k*te zentrale Moment* z_X^k definiert als

$$\begin{aligned} m_X^k &= \mathbf{E}\left[X^k\right] \\ z_X^k &= \mathbf{E}\left[(X - \mathbf{E}[X])^k\right] . \end{aligned}$$

Das erste Moment ist der Erwartungswert von X und wird manchmal mit μ bezeichnet. Das zweite zentrale Moment heißt auch *Varianz* und wird mit $\mathbf{var}[X]$ oder σ_X^2 bezeichnet. Die Größe σ_X heißt auch *Standardabweichung*. ◇

A.23 Lemma. $\mathbf{var}[X] = m_X^2 - \mu_X^2 = \mathbf{E}\left[X^2\right] - \mathbf{E}[X]^2$.

 © Th. Worsch 2000-2015