Rheinisch-Westfälische Technische Hochschule Aachen
Lehr- und Forschungsgebiet für Theoretische Informatik
Prof. Dr. P. Rossmanith

Fingerprinting Techniques

Seminar Ergebnisprüfung
WS 2003/2004

Christoph Bürger
Matrikelnummer: 228715

Inhaltsverzeichnis

Zusammenfassung

Beim Finterprinting wird versucht die Korrektheit komplexer Programme dadurch zu überprüfen, dass man nicht dass komplette Ergebnis eines Programms kontrolliert, sondern von diesem einen Fingerprint nimmt und diese vergleicht. Der Vergleich der Fingerprints sollte dabei deutlich einfacher sein, als das Testen des tatsächlichen Ergebnisses. Dieses Verfahren kann fehlerhaft sein. Es wird deshalb versucht eine Wahrscheinlichkeit $< \frac{1}{2}$ dafür zu erreichen, dass eine Untersuchung der Fingerprints die Korrektheit des Programms bescheinigt, obwohl das Programm fehlerhaft ist.

1 Einführung

Die Überprüfung der Korrektheit von Programmen ist nicht einfach, da die angewendeten Algorithmen z. T. sehr komplex sind, weshalb man sie nicht einfach analysieren kann. In dieser Arbeit wird das Fingerprinting vorgestellt, eine Methode, die dem Vergleich von Elementen eines Universums U dient. Allerdings kann es sehr aufwendig sein zwei Elemente eines großen Universums direkt zu vergleichen, deterministisch hat ein solcher Vergleich mindestens die Komplexität $\log |U|$, stattdessen wird ein Mapping aus dem Universum U in ein deutlich kleineres Universum V gewählt, dessen Elemente einfacher zu vergleichen sind, mit Komplexität $\log |V|$. Dieses Mapping wird so gewählt, dass die Wahrscheinlichkeit dafür, dass zwei Elemente aus U gleich sind, wenn ihre Bilder in V gleich sind ausreichend groß ist. Für ein Element $x \in U$ heißt das Bild von x in V *Fingerprint* von x.

Um die Korrektheit eines Programms zu überprüfen vergleicht man nun das berechnete Ergebnis mit dem korrekten Ergebnis. Um das korrekte Ergebnis zu erhalten müsste man allerdings naive Algorithmen einsetzen, die zuverlässig, aber sehr langsam sind. Um Laufzeit zu sparen benutzt man randomisierte Algorithmen, die die Fingerprints benutzen, um die Korrektheit des Ergebnisses zu verifizieren. So versucht man z. B. die Korrektheit der Matrix-Multiplikation $A \cdot B = C$ für A, B, C $n \times n$-Matrizen, die in Abschnitt 2 genauer vorgestellt wird, darüber zu verifizieren, dass man einen Zufallsvektor auf beiden Seiten von rechts an die Matrizen multipliziert und die Ergebnisse dann vergleicht, statt den naiven $O(n^3)$ Algorithmus zu verwenden. Diese Algorithmen werden so gestaltet, dass die Wahrscheinlichkeit dafür, dass sie ein Programm falsch einschätzen kleiner $\frac{1}{2}$ ist.

Während der gesamten Arbeit wird über einem nicht näher spezifizierten Feld $\mathbb{F}$ gerechnet, insbesondere wird nicht spezifiziert, ob das Feld endlich oder unendlich ist. Im verwendeten Modell zur Berechnung der Komplexität eines Programms hat jede Feldoperation (Addition, Subtraktion, Multiplikation, Division, Vergleich, Wahl eines Zufallelements) dieselben Kosten.

Im folgenden wird nun eine Fingerprinting-Technik für die Matrix-Multiplikation vorgestellt, die zuvor schon angedeutet wurde. Anschließend wird eine Methode zur Verifikation der Identität von Polynomen vorgestellt. Die Verifikation von perfekten Matchings in Graphen wird ebenfalls Bestandteil dieser Arbeit sein. Danach folgen Methoden zur Verifikation der Gleichheit von String und zum Pattern Matching. Ein Vergleich der vorgestellten Fingerprinting-Techniken bildet dann mit einer Zusammenfassung und einem kurzen Ausblick den Abschluss dieser Arbeit.

2 Matrix Multiplikation

In diesem Abschnitt wird ein Verfahren zur Verifikation der Korrektheit eines Programms zur Matrix-Multiplikation vorgestellt. Es wurde von Freivalds erstmals in [1] beschrieben und trägt deshalb auch den Beinamen „Freivalds' Technik".

Das Problem, das sich hier stellt besteht darin, dass zwei $n \times n$-Matrizen momentan bestenfalls in Laufzeit $O(n^{2,376})$ multipliziert werden können, was eine deutliche Verbesserung gegenüber dem naiven Ansatz in Zeit $O(n^3)$ ist. Der besere Algorithmus ist aber sehr kompliziert, und die Korrektheit einer Implementierung dieses Algorithmus muss bewiesen werden. Da Programm-Verifikation sehr kompliziert ist, wird hier versucht die Korrektheit der Ausgabe des Algorithmus zu bestätigen. Gegeben seien nun die Matrizen $A, B, C \in \mathbb{F}^{n \times n}$ mit dem Ziel zu beweisen, dass $AB = C$ gilt. Die Verwendung des naiven Algorithmus würde zwar zum korrekten Ergebnis führen, was aber den Vorteil des schnelleren Algorithmus ausgleichen würde, man könnte also direkt den naiven Algorithmus anwenden. Das im folgenden vorgestellte randomisierte Verfahren nutzt aus, dass man das Produkt von A und B nicht erneut berechnen, sondern nur seine Äquivalenz zu C beweisen muss. Es bietet eine Möglichkeit, um in $O(n^2)$ zu beweisen, dass das berechnete Ergebnis korrekt ist.

Der Algorithmus wählt zunächst einen Zufallsvektor $r \in \{0,1\}^n$. Dabei sind 0 und 1 die neutralen Elemente der Addition und der Multiplikation im Feld $\mathbb{F}$. Jedes Element von r wird gleichverteilt über der Menge $\{0,1\}$ gewählt. Nun berechnet man $x = Br$, $y = Ax = ABr$ und $z = Cr$ in Zeit $O(n^2)$. Es ist klar, dass wenn $AB = C$, dann auch $y = z$. Nun ist noch zu beweisen, dass für $AB \neq C$ die Wahrscheinlichkeit für $y \neq z$ wenigstens $\frac{1}{2}$ ist, denn im Fall $AB \neq C$ und $y = z$ verursacht der Algorithmus eine fehlerhafte Ausgabe.

Theorem 2.1 *Seien $A, B, C \in \mathbb{F}^{n \times n}$, mit $AB \neq C$. Für ein zufällig, gleichverteilt gewähltes $r \in \{0,1\}^n$ gilt dann*

$$Pr[ABr = Cr] \leq \frac{1}{2}.$$

Beweis: Sei $D = AB - C$, dann wissen wir, dass D nicht die Nullmatrix ist. Wir möchten die Wahrscheinlichkeit für $y = z$ bzw. die Wahrscheinlichkeit für $Dr = 0$ begrenzen. O. B. d. A. können wir Annehmen, dass die erste Zeile von D einen Eintrag ungleich 0 hat und dass alle Einträge ungleich 0 in dieser Zeile vor denen stehen, die gleich 0 sind. Sei d der Vektor, der die Einträge der ersten Zeile von D enthält. Weiterhin sei angenommen, dass die ersten $k > 0$ Einträge in d ungleich 0 sind. Nun betrachten wir die Wahrscheinlichkeit dafür, dass das Produkt von d und r ungleich 0 ist. Da der erste Eintrag von Dr exakt $d^T r$ entspricht, bringt uns dies zu einer unteren Schranke für die Wahrscheinlichkeit, dass $y \neq z$. Das Produkt $d^T r = 0$ genau dann, wenn

$$r_1 = \frac{-\sum_{i=2}^{k} d_i r_i}{d_1}. \tag{1}$$

Wir beziehen nun das Prinzip der verzögerten Entscheidungen und die Annahme, dass alle anderen Elemente von r vor r_1 gewählt wurden ein. Dann ist die rechte Seite von Formel 1 auf einen festen

Wert $v \in \mathbb{F}$ fixiert. Da r_1 gleichverteilt aus einer Menge der Größe 2 gewählt wird, ist die Wahrscheinlichkeit dafür, dass es gleich v ist höchstens $\frac{1}{2}$. **q.e.d.**

Somit wird in Zeit $O(n^2)$ das Problem der Verifikation der Matrix-Multiplikation auf das der Verifikation der Gleichheit zweier Vektoren reduziert, was in Zeit $O(n)$ gemacht werden kann. Damit ergibt sich eine Gesamtlaufzeit von $O(n^2)$ für diesen Monte-Carlo Algorithmus. Die Fehlerwahrscheinlichkeit kann auf $\frac{1}{2^k}$ reduziert werden, indem man das Verfahren k-mal unabhängig anwendet.

3 Identität von Polynomen

Im vorliegenden Abschnitt wird ein Fingerprint-Verfahren zur Verifikation von Verfahren zur Multiplikation von Polynomen vorgestellt. Anschließend wird dieses Verfahren zum Test der Identität von implizit gegebenen multivariaten Polynomen verwendet, was am Beispiel der Vandermonde-Matrix gezeigt wird. Dies führt zur Verallgermeinerung von „Freivalds' Technik" auf mulitvariate Polynome und schliellich zum Schwartz-Zippel Theorem.

Für den Fall der Poynom-Mulitplikation seien drei Polynome $P_1(x), P_2(x), P_3(x) \in \mathbb{F}[x]$ gegeben. Ziel ist es nun zu beweisen, dass $P_1(x) \times P_2(x) = P_3(x)$ gilt. Dazu sei angenommen, dass die Polynome $P_1(x)$ und $P_2(x)$ einen Grad kleiner oder gleich n haben. Dann gilt auf jden Fall, dass $Grad(P_3(x)) \leq 2n$. Für die Laufzeit des Fingerprint-Verfahrens macht man sich zu Nutze, dass ein Polynom in Zeit $O(n)$ in einem festen Punkt ausgewertet werden kann, die explizite symbolische Berechnung aber selbst durch Fourier-Transformation höchstens in Zeit $O(n \log n)$ geschehen kann.

Das eigentliche Verfahren besteht nun darin, dass man eine Menge $\mathbb{S} \subseteq \mathbb{F}$ mit Mächtigkeit wenigstens $2n + 1$ wählt. Aus dieser Menge wählt man nun gleichverteilt, zufällig ein Element $r \in \mathbb{S}$ aus und berechnet $P_1(r)$, $P_2(r)$ und $P_3(r)$ in Zeit $O(n)$. Wenn nun $P_1(r)P_2(r) = P_3(r)$ gilt, so wird auch die Formel $P_1(x) \times P_2(x) = P_3(x)$ als korrekt angesehen. Ansonsten wird diese als falsch angesehen. Der Algorithmus versagt nur, wenn trotz der Identität an der Stelle r die Formel für x im Allgemeinen nicht erfüllt ist. Nun sei $Q(x) = P_1(x)P_2(x) - P_3(x)$ definiert. Es ist klar, dass $Q(x) = 0$ genau dann gilt, wenn das Produkt richtig ist.

Nun wird gezeigt, dass wenn $Q(x) \not\equiv 0$, so gilt auch mit hoher Wahrscheinlichkeit, dass $Q(r) = P_1(r)P_2(r) - P_3(r) \neq 0$. Aus der Algebra ist bekannt, dass $Q(x)$ höchstens $2n$ verschiedene Wurzeln haben kann. Solange nicht $Q \equiv 0$ gilt, gibt es also nur $2n$ mögliche Wahlen von $r \in \mathbb{S}$ für die $Q(r) = 0$ gilt. Damit ergibt sich für den vorgestellten Algorithmus eine Fehlerwahrscheinlichkeit von $\frac{2n}{|\mathbb{S}|}$ und es ist klar, dass die Fehlerwahrscheinlichkeit durch Mehrfachanwendung des Algorithmus bzw. durch eine entsprechende Wahl von $\mathbb{S}$ verkleinert werden kann. Für den Fall, dass $\mathbb{F}$ unendlich ist kann man die Fehlerwahrscheinlichkeit auf 0 reduzieren, indem man r aus dem gesamten Bereich von $\mathbb{F}$ wählt, was allerdings unendlich viele Zufallsbits benötigen würde. Eine deterministische Variante dieses Algorithmus würde darin bestehen alle Elemente aus $\mathbb{S}$ einmal zu testen. Dies würde allerdings nur in Zeit $\Theta(n \log^2 n)$ funktionieren, was langsamer ist als die Zeit, die man benötigt, um $P_1(x)$ und $P_2(x)$ direkt zu multiplizieren.

Das vorgestellte Verfahren ist nicht nur für die Verifikation der Polynommulitplikation geeignet, sondern auch, um die Identität zweier Polynome zu überprüfen. Für explizit gegebene Polynome ist dieses Verfahren allerdings unnötig, da man diese ohnehin in Zeit $O(n)$ vergleichen kann, indem man die Koeffizienten vergleicht. Der randomisierte Algorithmus hat also dieselbe Laufzeit wie ein direkter Vergleich der Koeffizienten. Das randomisierte Verfahren hat allerdings Vorteile, wenn die Polynome nur implizit, quasi als Black-Box, gegeben sind und man keinen direkten Zugriff aus die Koeffizienten hat bzw. wo die Berechnung der Koeffizienten sehr hohe Kosten verursachen würde.

Hierzu sei als Beispiel die Berechnung der Determinante einer symbolischen Matrix erwähnt. Dieses Problem kann man als Identität von multivariaten Polynomen ansehen. Es sei zunächst daran erinnert, dass für eine $n \times n$-Matrix M die Determinate folgendermaßen definiert ist:

$$\det(M) = \sum_{\pi \in \mathbb{S}_n} = sgn(\pi) \prod_{i=1}^{n} M_{i,\pi(i)}.$$

Dabei ist $\mathbb{S}_n$ die symmetrische Gruppe der Permutationen der Größe n und $sgn(n)$ ist die Signatur der Permutation π. Zur Erinnerung: $sgn(\pi) = (-1)^t$ mit t ist die Anzahl paarweiser Elementvertauschungen, die benötigt werden, um die Identität in die Permutation π umzuwandeln. Wenn die Matrixeinträge M_{ij} explizit gegeben sind, so kann die Determinante in Polynomzeit berechnet werden, auch wenn sie $n!$ Terme enthält.

Definition 3.1 *Die* Vandermonde Matrix $M(x_1, \ldots, x_n)$ *ist in* $x_1, \ldots x_n$ *mit* $M_{ij} = x_i^{j-1}$ *definiert. Somit also:*

$$M = \begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_1^2 & \ldots & x_2^{n-1} \\ & & & \cdot & \\ & & & \cdot & \\ & & & \cdot & \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{pmatrix}.$$

Die Vandermondsche Identität besagt nun, dass für diese Matrix M gilt, dass
$\det(M) = \prod_{j<i}(x_i - x_j)$. Ziel ist es nun diese Identität zu verifizieren. Das Problem hierbei besteht in der Berechnung der Determinante, die $n!$ Terme enthält. Man kann das Problem dieses Beweises auch in der Form der Identität von zwei Polynomen $Q(x_1, \ldots, x_n) = \det(M) - \prod_{y<i}(x_i - x_j) = 0$ formulieren. Entsprechend der Idee von „Freivalds' Technik" wählt man nun für jedes x_i einen Zufallswert, setzt diese in $Q(x_1, \ldots, x_n)$ ein und prüft das Ergebnis auf Identität zu 0. Dieser Wert für Q ist nicht zu komplex zu berechen, da die Determinante in einem bestimmten Punkt in Polynomzeit berechenbar ist.

Hieran sieht man, dass man „Freivalds' Technik" auch auf den Fall von mulitvariaten Polynomen erweitern kann. Vor dem wichitgen Theorem über diese Verallgemeinerung, sei daran erinnert, dass für ein multivariates Polynom $Q(x_1, \ldots, x_n)$ gilt, dass der Grad eines Terms von Q der Summe der Exponenten der Variablen des Terms entspricht. Der *absolute Grad* des Polynoms Q ist als Maximum der Grade der Terme des Polynoms definiert.

Theorem 3.1 *(Schwartz-Zippel Theorem) Sei $Q(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n]$ ein multivariates Polynom mit absolutem Grad d. $\mathbb{S} \subseteq \mathbb{F}$ und seien $r_1, \ldots, r_n$ unabhängig voneinander gleichverteilt aus $\mathbb{S}$ gewählte Zufallswerte.*
Dann gilt:

$$Pr[Q(r_1, \ldots, r_n) = 0 | Q(x_1, \ldots, x_n) \not\equiv 0] \leq \frac{d}{|\mathbb{S}|}.$$

Beweis: Der Beweis wird über die Anzahl n der Variablen geführt. Der Basisfall $n = 1$ bezieht sich auf ein univariates Polynom $Q(x_1)$ vom Grad d. Für diesen Fall haben wir bereits im ersten Teil dieses Abschnitts gesehen, dass wenn $Q(x_1) \not\equiv 0$, so ist die Wahrscheinlichkeit für $Q(r_1) = 0$ höchstens $\frac{d}{|\mathbb{S}|}$. Nun sei angenommen, dass die Induktionsbehauptung für multivariate Polynome mit höchstens $n-1$ Variablen wahr ist, $n > 1$.

Nun betrachtet man das Polynom $Q(x_1, \ldots, x_n)$ und klammert x_1 aus, wodurch man

$$Q(x_1, \ldots, x_n) = \sum_{i=0}^{k} x_1^i Q_i(x_2, \ldots, x_n),$$

mit $k \leq d$ dem größten Exponenten von x_1 in Q erhält. Es sei angenommen, dass x_1 so Einfluss auf Q hat, dass $k > 0$ gilt. Durch die Wahl von k ist der Koeffizient $Q_k(x_2, \ldots, x_n)$ von x_1^k ungleich 0. Da der absolute Grad von Q_k höchstens $d - k$ ist, impliziert die Induktionsbehauptung, dass die Wahrscheinlichkeit für $Q_k(r_2, \ldots, r_n) = 0$ höchstens $\frac{d-k}{|\mathbb{S}|}$ ist.

Sei nun $Q_k(r_2, \ldots, r_n) \neq 0$. Nun betrachtet man das folgende univariate Polynom:

$$q(x_1) = Q(x_1, r_2, r_3, \ldots, r_n) = \sum_{i=0}^{k} x_1^i Q_i(r_2, \ldots, r_n).$$

Das Polynom $q(x_1)$ hat Grad k und ist ungleich 0, denn der Koeffizient von x_1^k ist $Q_k(r_2, \ldots, r_n)$. Der univariate Fall impliziert nun, dass die Wahrscheinlichkeit, dass $q(r_1) = 0$ gilt höchstens $\frac{k}{|\mathbb{S}|}$ ist.

Dadurch hat man nun die folgenden Ungleichungen bewiesen.

$$Pr[Q_k(r_2, \ldots, r_n) = 0] \leq \tfrac{d-k}{|\mathbb{S}|};$$
$$Pr[Q(r_1, r_2, \ldots, r_n) = 0 | Q_k(r_2, \ldots, r_n) \neq 0] \leq \tfrac{k}{|\mathbb{S}|}.$$

Es ergibt sich nun, dass die Wahrscheinlichkeit für $Q(r_1, \ldots, r_n) = 0$ nichts anderes als die Summe dieser zwei Wahrscheinlichkeiten, also $\frac{d}{|\mathbb{S}|}$, ist. **q.e.d.**

Das randomisierte Verfahren zum Beweis der Identität von Polynomen hat nur ein potenzielles Problem, denn wenn auf unendlichen Mengen gerechnet wird, so können sehr große Zwischenergebnisse entstehen, die die Berechnung sehr erschweren. Dieses Problem kann man zumindest im ganzzahligen Bereich umgehen, indem man eine „kleine" Primzahl p wählt und alle Berechnungen modulo dieser Primzahl durchführt. Dadurch wird die Berechnung vereinfacht und die Fehlerwahrscheinlichkeit wird nicht verändert.

In diesem Abschnitt ist ein randomisiertes Verfahren zur Verifikation der Identität von uni- und multivariaten Polynomen vorgestellt worden. Zunächst wurde es zum Beweis der Polynommultiplikation im univariaten Fall eingesetzt und später dann, mit identischer Technik auf den multivariaten Fall erweitert. Zuletzt wurde die Fehlerwahrscheinlichkeit für den multivariaten Fall im Schwartz-Zippel Theorem konstatiert und bewiesen.

4 Perfekte Matchings in Graphen

Die Methode, die zum Beweis der Vandermondschen Identität im Abschnitt 3 verwendet wurde, wird im folgenden Abschnitt angewendet, um die Existenz eines perfekten Matchings in einem bipartiten Graphen zu beweisen. Dabei wird, entsprechend der Beschreibung in [2], Edmonds Theorem im Mittelpunkt stehen, da dieses Theorem den Zusammenhang zwischen den perfekten Matchings in Graphen und der Determinante einer quadratischen Matrix herstellt.

Zur Einführung in das Verfahren sei zunächst noch einmal definiert, was ein Matching, und insbesondere ein perfektes Matching, ist. Dabei betrachtet man einen bipartiten Graphen $G(U, V, E)$ mit unabhängigen Knotenmengen $U = \{u_1, \ldots, u_n\}$ und $V = \{v_1, \ldots, v_n\}$. Ein *Matching* ist dann eine Menge von Knoten $M \subseteq E$, für die gilt, dass jeder Knoten höchstens einmal in M vorkommt. Ein *perfektes Matching* ist ein Matching der Größe n. Wichtig ist es zu wissen, dass man ein Matching M in G als eine Permutation von U nach V ansehen kann. Man kann die perfekten Matchings in G sogar in eine 1-zu-1-Korrespondenz mit den Permutation in $\mathbb{S}_n$ setzen. Dabei ist das Matching, das zur Permutation $\pi \in \mathbb{S}_n$ korrespondiert durch das Paar $(u_i, v_{\pi(i)})$ für $1 \leq i \leq n$ gegeben ist.

Theorem 4.1 *(Edmonds Theorem) Sei A die folgendermaßen aus $G(U, V, E)$ entstehende $n \times n$ Matrix (Edmonds Matrix):*

$$A_{ij} = \begin{cases} x_{ij} & (u_i, v_j) \in E \\ 0 & (u_i, v_j) \notin E \end{cases} .$$

Definiere das multivariate Polynom $Q(x_{11}, x_{12}, \ldots, x_{nn}) = \det(A)$. Dann hat G genau dann ein perfektes Matching, wenn $Q \not\equiv 0$.

Beweis: Die Determinante von A ist gegeben durch

$$\det(A) = \sum_{\pi \in \mathbb{S}_n} sgn(\pi) A_{i,\pi(1)} A_{2,\pi(2)} \ldots A_{n,\pi(n)}.$$

Da jedes x_{ij} nur einmal in A auftaucht können sich in der Summe keine Terme gegenseitig eliminieren. Also ist die Determinante genau dann ungleich 0, wenn eine Permutation π existiert für die der korrespondierende Term in der Summe ungleich 0 ist. Dies tritt genau dann ein, wenn jeder der Einträge $A_{i,\pi(i)}$, für $1 \leq i \leq n$, ungleich 0 ist. Dies ist äquivalent dazu, dass es ein perfektes Matching in G gibt. **q.e.d.**

Mit dem Algorithmus aus Abschnitt 3 lässt sich nun einfach ein Verfahren entwickeln mit dem man einen bipartiten darauf prüfen kann, ob in ihm ein perfektes Matching enthalten ist, indem man die Determinante der Matrix auf Äquivalenz zu 0 prüft. Die Laufzeit des Algorithmus ist dabei durch Berechnung der Determinante dominiert, was ungefähr der Multiplikation zweier Matrizen entspricht. Es gibt Algorithmen, die in der Lage sind ein perfektes Matching in Zeit $O(m\sqrt{n})$, mit $m = |E|$, zu *konstruieren*. Da die Zeit zur Berechnung der Determinante $m\sqrt{n}$ für kleines m überschreitet, bringt das vogestellte Verfahren hier bestenfalls geringfügige Vorteile. Der vorgestellte Algorithmus hat allerdings Vorteile, wenn es um die Parallelisierung geht, weshalb seine Vorstellung nicht ohne Sinn ist.

Entsprechend [3] kann man auch ein Verfahren für allgemeine, nicht zwingend bipartite, Graphen entwickeln, das ein entsprechendes Ergebnis liefert. Dieses soll hier nicht näher vertieft werden, es sei aber darauf hingewiesen, dass für allgmeine Graphen der „Satz von Tutte" das Gegenstück zum hier verwendeten Theorem von Edmonds ist.

In diesem Abschnitt ist ein Verfahren beschrieben wurden, mit dem man testen kann, ob ein bipartiter Graph ein perfektes Matching enthält. Durch Edmonds Theorem wurde die Existenz eines perfekten Matchings dabei auf die Äquivlanez der Determinante einer Matrix mit 0 zurückgeführt. Dieser Test konnte dann mit dem Verfahren aus Abschnitt 3 gelöst werden.

5 Identität von Strings

Im folgenden Abschnitt wird ein randomisiertes Verfahren vogestellt, mit welchem man feststellen kann, ob zwei gegebene Strings (Bitvektoren) identisch sind oder nicht. Das Verfahren beruht im wesentlichen darauf, dass man die gegebenen Strings als Zahlen interpretiert und die Residuen dieser Zahlen modulo einer Primzahl p dann miteinander vergleicht.

Man stelle sich folgendes Szenario vor: *Alice* und *Bob* besitzen Kopien derselben Datenbank. Dabei sollen die Kopien der Datenbanken als Bitvektoren $(a_1, \ldots, a_n)$ bei Alice und $(b_1, \ldots, b_n)$ bei Bob dargestellt sein. Diese Kopien der Datenbanken müssen in regelmäßigen Abständen daraufhin getestet werden, ob sie noch gleich sind. Ein deterministischer Konsistenzcheck würde nun n Bits, also den kompletten Bitvektor der einen Seite, übertragen müssen, da ansonsten ein Angreifer das Verfahren zunichte machen könnte. Da die Kommunikation zwischen Alice und Bob, aber sehr teuer ist, sollte ein Verahren gefunden werden, dass bei möglichst geringer Fehlerwahrscheinlichkeit deutlich weniger als n Bits übertragen muss.

Ein randomisiertes Verfahren besteht nun darin, dass man die Strings in n-Bit Integer-Zahlen a und b umwandelt, so einen Fingerprint erstellt und diesen überträgt. Dabei seien die Integer-Zahlen, die statt der Bitvektoren verwendet werden wie folgt definiert:

$$a = \sum_{i=1}^{n} a_i 2^{i-1} \qquad b = \sum_{i=1}^{n} b_i 2^{i-1}.$$

Die Fingerprint-Funktion ist dann als $F_p(x) = x \bmod p$ für eine Primzahl p definiert. Nach dieser Umwandlung wird nur noch der Wert $F_p(a)$ von Alice an Bob übertragen. Bob vergleicht dann die Werte von $F_p(a)$ und $F_p(b)$. Wenn dann $F_p(a) = F_p(b)$ gilt, so geht man davon aus, dass die Datenbanken noch konsistent sind, ansonsten betrachtet man sie als ungleich. Wichtige Annahme dabei ist natürlich, ähnlich wie in den anderen Verfahren, dass wenn $a \neq b$, so auch $F_p(a) \neq F_p(b)$. Dadurch verringert sich die Anzahl der zu übertragenden Bits von n Bits auf $O(\log p)$. Diese Strategie hat den Nachteil, dass ein Angreifer sie für festes p leicht überlisten kann, da für gegebene Zahlen p und b viele a mit $a \equiv b(\bmod p)$ existieren. Eine Lösung hierfür besteht darin, dass man die Primzahl p zufällig wählt.

Zur Analyse der Fehlerwahrscheinlichkeit sei zunächst $\pi(k)$ für jedes k als die Zahl der unterschiedlichen Primzahlen kleiner k definiert. Das Primzahl-Theorem besagt nun, dass $\pi(k)$ ungefähr gleich $\frac{k}{\ln k}$ ist. Weiterhin sei $c = |a - b|$ definiert. Entsprechend [2] versagt der definierte Fingerprint nun nur in dem Fall, in dem $c \neq 0$ gilt und p teilt c. Wenn nun $N = 2^n$ definiert wird, so wissen wir, dass $c \leq N$ gilt.

Lemma 5.1 *Die Anzahl der verschiedenen Primteiler einer beliebigen Zahl kleiner als 2^n ist höchstens n.*

Beweis: Jede Primzahl ist größer als 1. Wenn N mehr als t verschiedene Primteiler hätte, so wäre $N \geq 2^t$. **q.e.d.**

Damit der Bereich aus dem die Primzahl p gewählt wird nicht zu groß wird, wählt man nun einen Schwelle τ, die größer als $n = \log N$ ist. Es gilt $\pi(\tau) \sim \frac{\tau}{\ln \tau}$. Von den Primzahlen kleiner τ können nun höchstens n Teiler von c sein und somit einen Fehler der Fingerprint-Funktion verursachen. Man wählt p nun aus dem Bereich kleiner τ, um F_p zu definieren. Dadurch braucht man nur noch $O(\log \tau)$ Kommunikationsbits. Eine gute Wahl für τ ist nun, wenn man $\tau = tn \log tn$ für ein großes t wählt. Nun folgt das nachfolgende Theorem direkt, indem man die Wahrscheinlichkeit durch die zufällige Wahl von p bestimmt.

Theorem 5.1

$$Pr[F_p(a) = F_p(b) | a \neq b] \leq \frac{n}{\pi(\tau)} = O\left(\frac{1}{t}\right).$$

Somit erhält man eine Fehlerwahrscheinlichkeit von $O\left(\frac{1}{t}\right)$ und die Anzahl der zu übertragenden Bits beträgt $O(\log t + \log n)$. Eine sehr gute Wahl für t ist $t = n$. Das einzige Problem, das hier vorkommt besteht darin, dass die Wahl einer zufälligen Primzahl nicht trivial ist.

In diesem Abschnitt ist ein Algorithmus zur Verifikation der Identität zweier Strings vorgestellt wurden. Dieses Verfahren beruhte auf der Interpretation der Strings als Binärzahl und ihrer Umwandlung in eine Integerzahl. Anschließend wurden nur noch die Intergerzahlen modulo einer zufällig gewählten Primzahl verglichen. Dieses Verfahren führte zu einer sehr geringen Fehlerwahrscheinlichkeit und verringert vor allen Dingen die Zahl der zu übertragenden Bits.

6 Pattern Matching

Im vorliegenden Abschnitt geht es, wie im vorhergehenden Abschnitt auch, um dem Vergleich von Strings. Allerdings geht es hier nicht darum komplette Texte auf ihre Identität zu untersuchen, sondern darum zu prüfen, ob ein gewisser Teiltext (ein Pattern) in einem Gesamttext vorkommt.

Zunächst soll das Pattern-Matching formal definiert werden. Gegeben sind der Gesamttext, in dem gesucht werden soll, in der Form $text := X = x_1 x_2 \dots x_n$ und der zu suchende Teiltext $pattern := Y = y_1 y_2 \dots y_m$. Sowohl $text$ als auch $pattern$ sind über einem festen Alphabet Σ, wobei im folgenden o. B. d. A. davon ausgegangen wird, dass $\Sigma = \{0, 1\}$ gilt. Eine weitere Voraussetzung ist natürlich, dass $m \leq n$ gilt. $pattern$ kommt in $text$ vor, wenn ein $j \in \{1, 2, \dots, n - m + 1\}$ existiert, so dass für $1 \leq i \leq m$, $x_{j+i-1} = y_i$ gilt. Man definiert nun $X(j) = x_j x_{j+1} \dots x_{j+m-1}$ als Sub-String der Länge m in X, der an Position j beginnt. Das Pattern Y kommt dann in $text$ vor, wenn ein j exisitiert mit $1 \leq j \leq n - m + 1$ und $Y = X(j)$. Da diese Bedingung für mehrere j erfüllt sein kann, wählt man immer das kleinste j, für das $X(j) = Y$ gilt, um die Lösung eindeutig zu machen.

Ein Brute-Force-Verfahren würde nun jedes mögliche $X(j)$ mit Y vergleichen. Dieser Algorithmus hat eine Laufzeit von $O(nm)$. Es existieren auch gute deterministische Algorithmen mit einer Laufzeit von $O(n + m)$. Hier wird nun ein randomisierter Algorithmus vorgestellt, der ebenfalls über einen Fingerprint untersucht, ob das Pattern im Text vorkommt. Von diesem Algorithmus werden mehrere Varianten, genauer gesagt ein Monte-Carlo Algorithmus und zwei Las-Vegas Algorithmen, vorgestellt, die das Problem lösen. Die Fingerprint-Funktion F wird nun nach [2] entsprechend derjenigen aus Abschnitt 5 folgendermaßen definiert. Für jeden String $Z \in \{0, 1\}^m$, interpretiert man Z als m-bit Integer. Dann definiert man $F_p(Z) = Z \bmod p$. Dabei ist eine Primzahl, die gleichverteilt unter allen Primzahlen kleiner einer Schwelle τ gewählt wird. Das Verfahren interpretiert nun sowohl Y als auch $X(j)$ als m-bit Integer und vergleicht dann $F_p(Y)$ und $F_p(X(j))$. Wenn die Fingerprints identisch sind, so kommt ein Match vor, ansonsten nicht. Ein Fehler tritt genau dann auf, wenn für den Fall $Y \neq X(j)$ dennoch die Fingerprints identisch sind. Aus den Diskussion in Abschnitt 5 folgt unmittelbar, dass für die Fehlerwahrscheinlichkeit folgendes gilt:

$$Pr[F_p(Y) = F_p(X(j)) | Y \neq X(j)] \leq \frac{m}{\pi(\tau)} = O\left(\frac{m \log \tau}{\tau}\right).$$

Die Fehlerwahrscheinlichkeit für einen der höchstens n Werte von j ist $O\left(\frac{nm \log \tau}{\tau}\right)$. Wählt man nun $\tau = n^2 m \log n^2 m$, so ergibt sich eine Fehlerwahrscheinlichkeit von $O\left(\frac{1}{n}\right)$. Die Monte-Carlo Version des Algorithmus vergleicht nun die Fingerprints aller $X(j)$ mit dem von Y und gibt das erste j aus für das ein Match auftritt. Bevor nun die Las-Vegas Versionen dieses Verfahren vorgestellt wird, soll noch die Laufzeit dieses Algorithmus dargestellt werden. Für $1 \leq j \leq n - m + 1$ gilt $X(j + 1) = 2[X(j) - 2^{m-1} x_j] + x_{j+m}$. Daraus folgt nun

$$F_p(X(j + 1)) = 2[F_p(X(j)) - 2^{m-1} x_j] + x_{j+m} \bmod p.$$

Für ein gegebenens $X(j)$ kann $X(j + 1)$ mit $O(1)$ Kosten berechnet werden, woraus die Gesamtlaufzeit von $O(n + m)$ folgt. Aus den Ausführungen zur Fehlerwahrscheinlichkeit und der Laufzeit folgt nun unmittelbar das folgende Theorem.

Theorem 6.1 *Der Monte-Carlo Algorithmus für das Pattern Matching hat eine Laufzeit von* $O(n + m)$ *und hat eine Fehlerwahrscheinlichkeit von* $O\left(\frac{1}{n}\right)$.

Die Las-Vegas Algorithmen gehen prinzipiell ganauso vor wie der Monte-Carlo Algorithmus, allerdings vergleichen sie die Ursprungstexte $X(j)$ und Y miteinander nachdem sie einen Match per Fingerprint festgestellt haben. Wenn bei diesem direkten Vergleich ein false-match festgestellt wird, so bricht der erste Algorithmus ab und startet den Brute-Force Algorithmus mit Laufzeit $O(nm)$. Dieses Las-Vergas Verfahren arbeitet fehlerfrei und hat eine erwartete Laufzeit von $O((n+m)(1-\frac{1}{n}) + nm(\frac{1}{n})) = O(n+m)$. Allerdings hat dieses Verfahren im Einzelfall eine hohe Varianz in der Laufzeit, da das Ausführen, der Brute-Force Methode die Laufzeit sehr negativ beeinflussen kann. Die zweite Las-Vegas Methode bricht ebenfalls ab, falls beim direkten Vergleich ein false-match festgestellt wurde, allerdings startet sie sich dann selbst erneut mit einer anderen Wahl für die Primzahl p. Dabei ist die Wahrscheinlichkeit für mehr als t Neustarts $\leq \frac{1}{n^t}$. Dieses Verfahren hat eine deutlich kleinere Varianz der Laufzeit, da hier kein Algorithmus mit deutlich höhere Laufzeit als „Ersatzmethode" genutzt wird.

In diesem Abschnitt wurde ein Fingerprint-Verfahren zum Pattern Matching vorgestellt. Es benutzt prinzipiell dieselbe Fingerprintfunktion wie sie auch für die Überprüfung der Identität von Strings verwendet wird, sie wird hier allerdings auf Sub-Strings angewendet. Für diese Fingerprint-Fuktion wurden drei verschiedenen Algorithmen vorgestellt, die das Auftreten eines Matches prüfen, ein Monte-Carlo Algorithmus mit Fehlerwahrscheinlichkeit $O\left(\frac{1}{n}\right)$ und zwei Las-Vegas Algorithmen. Dabei spielt die Varianz der Laufzeit die entsccheidende Rolle welchen Algorithmus man wählt.

7 Zusammenfassung und Ausblick

Dieser Abschnitt soll noch einmal einen Überblick über die verwendeten Fingerprinting-Techniken geben, und auch eine hier nicht verwendete Technik als kurzen Ausblick vorstellen.

Die Techniken werden hier anhand vom Vergleich von Bit-Vektoren gegenübergestellt. Dabei sind die zwei Bit-Vektoren $a = (a_1, \ldots, a_n)$ und $b = (b_1, \ldots, b_n)$ mit $a_i, b_i \in \Sigma$ mit Σ endlich gegeben. Das Alphabet Σ ist dabei mit der Zahlenmenge $\Gamma = \{0, 1, \ldots, k-1\}$ mit $k = |\Sigma|$ kodierbar. Die Bit-Vektoren a und b werden dann in Polynome $A(z) = \sum_{i=0}^{n-1} a_i z^i$ und $B(z) = \sum_{i=0}^{n-1} b_i z^i$ mit Integer-Koeffizienten und Grad $\leq n$ umgewandelt. Dabei gilt natürlich $a = b \Leftrightarrow A(z) = B(z)$.

In der ersten zu betrachtenden Technik wird nun zunächst eine feste Primzahl p mit $p > 2n$ und $p > k$ gewählt. $A(z)$ und $B(z)$ werden dann als Polynome über $\mathbb{Z}_p$ betrachtet und $\Gamma \subseteq \mathbb{Z}_b$. Dabei wird die Arithmetik modulo p ungleiche Polynome nicht gleich machen, also das Ergebnis nicht negativ beeinflussen. Der Fingerprint der Polynome besteht nun darin, dass man ein zufälliges $r \in \mathbb{Z}_p$ und berechnet $A(r)$ und $B(r)$. Es gilt nun, dass $a = b \Rightarrow A(z) = B(z) \Rightarrow A(r) = B(r)$. Für $a \neq b$ folgt nun, dass $Pr[A(r) = B(r) | a \neq b] \leq \frac{n}{p} \leq \frac{1}{2}$ für ein entsprechend gewähltes p gilt. Für $k = 2$ und $p = O(n)$ kann man dies als Reduktion des Problems des Vergleichs von n-bit Zahlen auf den Vergleich von $O(\log n)$-bit Zahlen betrachten. Zu diesem Verfahren kann man zusammenfassend

sagen, dass man hier den Körper, über dem gerechnet wird, fixiert und anschließend einen zufälligen Berechnungspunkt wählt, der zum Erstellen des Fingerprints dient. Diese Technik wurde zum Beispiel bei der Matrix-Multiplikation in Abschnitt 2 und bei der Identität von Polynomen in Abschnitt 3 benutzt.

Die zweite zu betrachtende Technik geht einen anderen Weg, denn hier wird der Punkt der Evaluierung fixiert und anschließend wird zufällig der Körper gewählt über dem gerechnet wird. Man wählt hier zunächst fest $z = 2$ und anschließend eine zufällige Primzahl q ausreichend klein. Die Fingerprints bestehen nun aus $A(2)$ und $B(2)$ über dem Körper $\mathbb{Z}_q$. Diese Technik reduziert den Vergleich von n-bit Zahlen auf den von $\log n$-bit Zahlen und hat in manchen Bereichen Vorteile gegenüber der ersten vorgestellten Technik und findet z. B. Anwendung beim Pattern Matching.

Eine dritte Fingerprinting-Technik, die in keinem der vorrangehenden Abschnitte Verwendung fand, setzt nun zuerst eine Zahl $k = 2$. Die Vektoren a und b werden als n-bit Integer Zahlen interpretiert. Nun fixiert man eine Primzahl $p > 2^n$ und wählt ein zufälliges Polynom $P(z) \in \mathbb{Z}_p[z]$. Als Fingerprints bestimmt man nun $P(a)$ und $P(b)$ über $\mathbb{Z}_p$. Dann bestimmt man die Reste der Zahlen modulo einer Zahl nahe $\log n$. Dieses Verfahren ist die Kernidee hinter den so genannten *universal hash functions*, die hier nicht weiter vorgestellt werden.

Abschließend sei noch einmal kurz wiederholt was in dieser Arbeit behandelt wurde. Es wurden Fingerprint-Techniken für verschiedene Probleme vorgestellt. Dabei wurde zunächst in Abschnitt 2 anhand der Matrix-Multiplikation die Idee von Freivalds Technik eingeführt, welche später immer wieder Verwendung fand. Ebenfalls wurden Fingerprint-Techniken für die Prüfung der Identität von Polynomen, zur Prüfung der Identität von Strings und zur Lösung des Pattern-Matching Problems vorgestellt. Die dabei verwendeten Verfahren waren sehr ähnlich, da auch die zu untersuchenden Strukturen sehr eng verwandt sind. Auch der Beweis der Existenz eines perfekten Matchings in einem Graphen ist eng verwandt mit der Überprüfung der Identität von Polynomen, da sie durch den Bezug zur Determinante einer symbolischen Matrix auf die Überprüfung der Identität zweier Polynome zurückgeführt wurde. Insgesamt kann man sagen, dass die hier vorgestellten Techniken schon sehr gut sind und in guter Laufzeit bei geringer Fehlerwahrscheinlichkeit funktionieren.

Literatur

[1] Rusins Freivalds. Fast probabilistic algorithms. Technical report, Computing Center, Latvian University Riga, USSR, 1979.

[2] Rajeev Motwani and Prabhakar Rhaghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[3] Angelika Steger. Vorlesung Randomisierte Algorithmen, 2003.