Fingerprinting

Markus Jehle

22.1.2003

Zusammenfassung

Fingerprinting ist ein Sammelbegriff für verschiedene probabilistische Verfahren, die durch Vereinfachung an einer Anwendung beteiligter Größen für eine erhebliche Laufzeitverbesserung im Gegensatz zu deterministischen Lösungsansätzen sorgen.
In dieser Arbeit werden einige Fingerprintingverfahren anhand von Anwendungsbeispielen wie der *Verifikation von Gleicheit (Matrizen, Polynomen,...)*, dem *Auffinden von "Perfekten Matchings" in Graphen* oder *Pattern Matching in Strings* erklärt.

1 Einleitung

Schon 1901, lange vor dem ersten Computer, führte unter anderem Francis Galton bei Scotland Yard eine Art Fingerabdruckverfahren ein. Dabei hat man sicher wenig an das gedacht, was heute in der Informatik im Zusammenhang mit Probabilistischen Algorithmen als Fingerprinting — ins Deutsche am besten zu übersetzten mit „Fingerabdruckverfahren“ — bezeichnet wird. Es lohnt sich aber vor dem Einstieg in die Materie einen kurzen Blick auf so ein „nichtinformatisches“ Fingerabdruckverfahren zu werfen, wie es zu Galtons Zeit ausgesehen haben könnte. Die Gemeinsamkeiten mit dem was man heute in der Informatik als Fingerprinting bezeichnet sind ziemlich weitreichend.
Von allen Straftätern werden die Fingerabdrücke genommen. Wenn am Ort eines Verbrechens Fingerabdrücke gefunden werden, kann man diese mit den bereits registrierten Fingerabdrücken Verdächtiger vergleichen um sie gegebenenfalls zu überführen.
Es war Anfang des letzten Jahrhunderts schon rein technisch utopisch die Fingerabdrücke aller Menschen zu nehmen — geschweige denn diese dann auch noch mit einem gefundenen Fingerabdruck zu vergleichen.
So beschränkte man sich auf die viel kleinere Menge der bereits einmal überführten Straftäter. Die Wahrscheinlichkeit einen Verbrecher anhand dieser Fingerabdruckkartei zu überführen ist so zwar nicht mehr 100%, aber das Verfahren war in dieser wesentlich kleineren Dimension auch schon Anfang des letzten Jahrhunderts realisierbar.

In diesem Verfahren steckt schon das komplette Grundprinzip des Fingerprintings wie es heute in Algorithmen eingesetzt wird. Folgende eher mathematische Erklärung des Begriffs wird dies deutlich machen.

2 Begriffserklärung

Zugrunde liegt eine Aufgabenstellung im Zusammenhang mit Elementen eines beliebig großen Universums U. Ein Beispiel für so eine Aufgabenstellung wäre die Überprüfung auf Gleichheit zweier Elemente x und y aus U. Ein deterministischer Algorithmus für diese Überprüfung hat mindestens eine Laufzeit von $O(log|U|)$.[1]
Die Idee des Fingerprintings besteht nun darin, eine Abbildung von U in ein viel kleineres Universum V einzuführen, so dass ein Algorithmus für die zugrundeliegende Aufgabenstellung in V immer noch mit möglichst großer Wahrscheinlichkeit ein (im Bezug auf U) korrektes Ergebnis liefert. Die Gleicheit zweier Elemente in V wäre dann mit $O(log|V|)$, also mit niedrigerer Laufzeit berechenbar als in U; die Wahrscheinlichkeit eines korrekten Ergebnisses im Bezug auf U wäre aber nicht mehr 100%.
Der neue Algorithmus ist somit nicht mehr deterministisch, sondern probabilistisch. Die Korrektheit unterliegt einer gewissen Wahrscheinlichkeit.
Als Fingerprints werden nun die Bilder der Abbildung von U nach V bezeichnet, mit denen die vereinfachte Berechnung durchgeführt wird.

3 Fingerprintingverfahren und ihre Anwendungen

3.1 Technische Vorbemerkungen

Den folgenden Ausführungen wird ein nicht näher spezifizierter Körper $\mathbb{F}$ zugrundegelegt. Ob der Körper endlich ist oder nicht spielt keine Rolle, da die folgenden Algorithmen immer mit endlichen Teilmengen operieren.
Laufzeitberechnungen wird das „uni-cost RAM model“ zugrunde gelegt. Das bedeutet, dass jede Berechnungsoperation mit höchstens polynomieller Laufzeit über einem Körper als „gleich-komplex“ angenommen wird. Das hat vor allem den Hintergrund, die Komplexitätsvergleiche zwischen Algorithmen zu vereinfachen. Ein diesbezüglich feineres Modell würde sich bei den hier behandelten Anwendungen nur in einem kleinen Faktor auswirken.

[1] Die Komplexität hängt von der Darstellung der Elemente von U ab. Werden die Elemente beispielsweise als Folge von Nullen und Einsen dargestellt, braucht man um ein Element von U darzustellen $log_2|U|$ Stellen. Um zwei Elemente zu vergleichen muss man dann bis zu $log_2|U|$ Stellen vergleichen.

3.2 „Freiwalds Verfahren“ und hierzu verwandte Verfahren

Bei Freiwalds Verfahren handelt es sich um einen einseitigen Monte-Carlo-Algorithmus mit Laufzeit $O(n^2)$, der im wesentlichen zwei Matrizen vergleicht. Wenn beide Matrizen explizit vorliegen, kann man dies natürlich auch mit dem "naiven" Vergleichsalgorithmus mit einer Laufzeit von $O(n^2)$ sogar auf deterministische Weise bewerkstelligen. Es gibt jedoch Situationen, in denen eine der beiden Matrizen nicht ohne erheblichen Aufwand berechenbar ist. Genau hier setzt Freiwalds Verfahren an.

3.2.1 Freiwalds Verfahren zur Verifikation einer Matrixmultiplikation

Der naive Matrixmultiplikationsalgorithmus berechnet das Produkt von zwei Matrizen mit Laufzeit $O(n^3)$. Es gibt auch schnellere Algorithmen wie etwa den von Strassen mit einer Laufzeit von $\sim O(n^{2,81})$. Der schnellste bisher gefundene hat die Laufzeit $\sim O(n^{2,376})$. Die schnelleren Algorihmen haben nun leider den Nachteil, dass sie sehr kompliziert sind.
Die Korrektheit solcher Algorihmen zu beweisen ist ein sehr schwieriges Thema. Eine Alternative könnte sein die Korrektheit so eines Algorithmus bzw. dessen korrekte Realisierung anhand seiner Ausgabe zu prüfen. Dies mit Hilfe eines andereren Matrixmuliplikationsalgorithmus zu tun würde die gewonnenen Laufzeitvorteile aber wieder zunichte machen. Freiwalds Verfahren gibt hierzu nun einen Algorithmus mit Laufzeit $O(n^2)$ an, der die korrekte Ausgabe einer Matrixmultiplikation mit einer begrezten Fehlerwahrscheinlichkeit prüft.

Algorithmus 1 (Freiwald) *Seien $A, B \in {}^n\mathbb{F}^n$. Sei $r \in \{0,1\}^n$ ein Vektor, dessen Komponenten unabhängig voneinander zufällig aus $\{0,1\}$ ausgewählt wurden.*

x:=Br
x:=Ax
y:=Cr
if x=y return true
else return false

□

Es ist klar, dass der Algorithmus *true* zurückliefert, wenn $AB = C$ ist, also wenn der Matrixmultiplikationsalgorithmus korrekt arbeitet. Ist dagegen $AB \neq C$ gibt es eine gewisse Wahrscheinlichkeit, dass der Algorithmus beim Vergleich der *Fingerabdrücke* fälschlicherweise true zurückliefert. Diese Wahrscheinlichkeit ist jedoch begrenzt, was folgender Satz formuliert:

Satz 1 (Freiwald) *Seien $A, B, C \in {}^n\mathbb{F}^n$ so dass $AB \neq C$. Dann gilt für alle zufällig ausgewählten $r \in \{0,1\}^n$, dass $Pr[ABr = Cr] \leq \frac{1}{2}$.*

Beweis: Es gilt nun die Wahrscheinlichkeit zu bestimmen, dass $ABr = Cr$ unter der Bdeingung $AB \neq C$.
$ABr = Cr \Leftrightarrow \underbrace{(AB - C)}_{=:D} r = 0.$
Wir wissen, dass D mindestens einen von Null verschiedenen Eintrag besitzt. O.B.d.A. sei $d_i (1 \leq i \leq n)$ die erste Zeile von D, in der ein von Null verschiedener Eintrag steht.
O.B.d.A. stehen alle k von Null verschiedenen Einträge in d_i vor den mit Null identischen.
$d^T r$ ist genau der i-te Eintrag in Dr.
Nun gilt: $d^T r = 0 \Leftrightarrow \underbrace{r_1 = \frac{-\sum_{i=2}^{n}(d_i r_i)}{d_1}}_{(*)}$
Wenn man unter Anwendung des Prinzips der *Aufgeschobenen Entscheidung*[2] annimmt, dass alle Werte von r vor r_1 festgelegt werden, hat die rechte Seite von $(*)$ einen festen Wert aus $\mathbb{F}$. Nun gibt es höchstens einen Wert für r_1, so dass $d^T r = 0$ ist. Da $r_1 \in \{0, 1\}$, also Element einer zweielementigen Menge ist, gilt:
$\Rightarrow Pr[ABr = Cr] \leq \frac{1}{2}$

□

Die Fehlerwahrscheinlichkeit kann man durch i Wiederholungen des Tests auf einen Wert $\leq (\frac{1}{2})^i$ senken. Selbst bei vielen Wiederholungen bleibt die Gesamtlaufzeit der Tests noch weit unter der Laufzeit einer Matrixmultiplikation.

3.2.2 Gleichheit von univariaten Polynomen

Die Idee von Freiwalds Verfahren kann man auch für die Verifikation der Gleichheit von Polynomen verwenden. Auch hier kann man Polynome wieder mit Hilfe des naiven Algorithmus vergleichen, indem man ihre Koeffizienten vergleicht. Aber auch hier gibt es wieder Fälle, in denen das nicht so einfach klappt. Ein erstes Beispiel dafür ist Algorithmus 2, ein einfacher probabilistischer Algorithmus zur Verifikation einer Multiplikation von zwei Polynomen. $P_1 \cdot P_2 = P_3$ durch Ausrechnen der linken Seite zu prüfen hat schon den Nachteil, dass das Ausrechnen von $P_1 \cdot P_2$ mindestens die Laufzeit $O(n \log n)$ hat, wobei n das Maximum der Grade von P_1 und P_2 ist. Alle drei Polynome an einer Stelle $r \in \mathbb{F}$ auszuwerten und dann nur dort die Gleichheit zu prüfen hätte nur Laufzeit $O(n)$. Dies ist die Basis für den folgenden Algorithmus.

[2]Die Idee des Prinzips der *Aufgeschobenen Entscheidung* besteht darin, anzunehmen, dass nicht alle zufälligen Entscheidungen im Voraus getroffen werden. Bei jedem Schritt des Prozesses werden nur die für den Algorithmus notwendigen zufälligen Entscheidungen getroffen.

Algorithmus 2 *Seien $P_1, P_2, P_3 \in \mathbb{F}[x]$.*
Sei n der höchste in P_1 und P_2 vorkommende Grad.
Sei $\mathbb{S} \subseteq \mathbb{F}$ eine Menge mit mindestens $2n+1$ Elementen.
Sei $r \in \mathbb{S}$ zufällig ausgewählt.

a:=$P_1(r) \cdot P_2(r)$
if $a = P_3(r)$ *return true*
else return false

□

Wichtig ist hier natürlich auch wieder die Beschränktheit der Fehlerwahrscheinlichkeit. Es ist klar, dass der Algorithmus bei $P_1 \cdot P_2 = P_3$ immer true zurückliefert. Für den anderen Fall liefert folgender Satz eine zu dem Satz über Freiwalds Verfahren sehr ähnliche Aussage.

Satz 2 *Seien $P_1, P_2, P_3 \in \mathbb{F}[x]$ so dass $P_1 \cdot P_2 \neq P_3$. Sei n der höchste in P_1, P_2 vorkommende Grad. Dann gilt für ein zufällig ausgewähltes $r \in \mathbb{S}$, dass $Pr[P_1(r) \cdot P_2(r) = P_3(r)] \leq \frac{2n}{|\mathbb{S}|}$.*

Beweisskizze (vom Prinzip ähnlich zum Beweis von Satz 1): Sei $Q = P_1 \cdot P_2 - P_3$ ein Polynom. Q hat höchstens den Grad $2n$. Der Algorithmus gibt fälschlicherweise true zurück, wenn $Q(r) = 0$.
Q hat höchstens $2n$ Nullstellen. Daher gibt es höchstens $2n$ verschiedene $r \in \mathbb{S}$, für die $Q(r) = 0$ gilt. Somit ist die Fehlerwahrscheinlichkeit von Algorithmus 2 höchstens $\frac{2n}{|\mathbb{S}|}$.

□

3.2.3 Gleichheit von multivariaten Polynomen

Ein weiterer Fall, in dem Polynome nicht so einfach zu vergleichen sind, ist z.B. folgendes Problem:
Sei $M(x_1, \ldots, x_n)$ eine Vandermondsche Matrix (d.h. $M_{ij} = x_i^{j-1}$, wobei die x_i Variablen sind). Für die Determinante einer solchen Vandermondschen Matrix gilt nun:

$$det(M) = \prod_{1 \leq j \leq n, 1 \leq i \leq j} (x_i - x_j).$$

Ein Beweis für so eine Formel ist sehr komplex und das Ausrechnen so einer symbolischen Matrix ist sehr aufwendig. Stattdessen könnte man das Prinzip aus dem vorhergehenden Abschnitt aufnehmen, für multivariate Polynome erweitern und so die Gleichheit obiger Formel prüfen.
Der dadurch erhaltene Algorithmus ist fast identisch mit Algorithmus 2. Man muss lediglich die beiden Seiten der zu prüfenden Formel an einem Punkt $(r_1, \ldots, r_n)$ auswerten und dann die Ergebnisse auf Gleichheit prüfen. Eine Aussage zur Fehlerwahrscheinlichkeit bei diesem Verfahren formuliert der folgende Satz:

Satz 3 (Zippel-Schwartz) *Sei $Q(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n]$ ein multivariates Polynom mit Gesamtgrad[3] d. Sei $\mathbb{S} \subseteq \mathbb{F}$ und seien $r_1, \ldots, r_n$ unabhängig und zufällig aus $\mathbb{S}$ ausgewählt.*
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: *Induktion über die Anzahl der Variablen n*
Induktionsanfang $n = 1$: klar (siehe Satz 2)
Induktionsschritt $n \to n+1$:
Sei $Q(x_1, \ldots, x_{n+1})$ ein Polynom d-ten Grades. Durch Ausklammern von x_1 erhält man

$$Q(x_1, \ldots, x_{n+1}) = \sum_{i=0}^{k} x_1^i Q_i(x_2, \ldots, x_{n+1})$$

wobei k der höchste in Q vorkommende Exponent von x_1 ist. Da es sich bei den verschiedenen Q_i um Polynome mit n Variablen handelt, gilt für sie die Induktionshypothese. Insbesondere für Q_k, das aufgrund der Wahl von k nicht Null ist, gilt

$$\underbrace{Pr[Q_k(r_2, \ldots, r_{n+1}) = 0] \leq \frac{d-k}{|\mathbb{S}|}}_{=:(*)}.$$

Nehme nun an, dass $Q_k(r_2, \ldots, r_{n+1}) \neq 0$. Betrachte folgendes univariates Polynom:

$$q(x_1) = Q(x_1, r_2, r_3, \ldots, r_{n+1}) = \sum_{i=0}^{k} x_1^i Q_i(r_2, \ldots, r_{n+1}).$$

Da $q(x_1)$ nicht Null ist gilt nach Induktionsanfang

$$\underbrace{Pr[Q(r_1, \ldots, r_{n+1}) = 0 | Q_k(r_2, \ldots, r_{n+1}) \neq 0] \leq \frac{k}{|\mathbb{S}|}}_{=:(**)}$$

Wenn man (*) und (**) in die Formel über bedingte Wahrscheinlichkeiten $Pr[a_1] \leq Pr[a_1 | \overline{a}_2] + Pr[a_2]$ einsetzt erhält man:

$$Pr[Q(r_1, \ldots, r_{n+1}) = 0 | Q(x_1, \ldots, x_{n+1}) \not\equiv 0] \leq \frac{d}{|\mathbb{S}|}$$

□

Ein Problem bei den Verfahren zur Verifikation von Polynomen kann sein, dass bei der Berechnung enorme Werte erreicht werden können. Das hängt vor allem mit der möglichen Unendlichkeit von $\mathbb{F}$ zusammen. Einen Lösungsansatz für dieses Problem bietet ein später erläutertes Fingerprintingverfahren.

[3] Der Gesamtgrad eines multivariaten Polynoms ist gleich dem des Summanden mit maximalem Grad. Der Grad eines Summanden ist definiert als Summe der Grade der Variablen, die er enthält.

3.2.4 Finden eines perfekten Matchings in Graphen

Den Abschluss des Kapitels über Polynom-Verifikations-Algorithmen soll eine wirklich erstaunliche Anwendung machen. Es handelt sich um das Auffinden eines „Perfekten Matchings“ in Graphen.

Definition 1 (Matching) *Sei $G(U, V, E)$ ein bipartiter Graph mit unabhängigen Punktmengen $U = \{u_1, \ldots, u_n\}$ und $V = \{v_1, \ldots, v_n\}$ der Größe n.*
Ein „Matching“ ist eine Menge von Kanten so dass jeder Punkt als Teil einer Kante höchstens einmal auftaucht.
Ein „perfektes Matching“ ist ein matching der Größe n.

Folgender Satz stellt den Zusammenhang zwischen einem *perfektem Matching* und Determinanten her.

Satz 4 (Satz von Edmond) *Sei A eine $n \times n$-Matrix, die man aus $G(U, V, E)$ wie folgt erhält:*

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

Das multivariate Polynom Q sei definiert als $Q(x_{11}, x_{12}, \ldots, x_{nn}) = det(A)$.
Dann gilt:

In G existiert ein "perfektes Matching" $\Leftrightarrow Q \not\equiv 0$.

Beweis: Die Determinante ist wie folgt definiert:

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

Da jede Unbekannte x_{ij} in A höchstens einmal vorkommt, können sich die einzelnen Summanden der Determinante nicht auslöschen.
Die Determinante von A ist genau dann nicht Null, wenn mindestens einer ihrer Summanden nicht Null ist.
$\Leftrightarrow$ In den von null verschiedenen Summanden steht anstelle jedes $A_{i,\pi(i)} (1 \leq i \leq n)$ ein Wert ungleich Null.
$\Leftrightarrow$ Zwischen jeder Punktekombination i und $\pi(i)$ ist eine Kante.
$\Leftrightarrow$ Es existiert ein perfektes Matching.

□

Mit den Ergebnissen aus dem Absatz über den Vergleich multivariater Polynome kann man jetzt einen probabilistischen Algorithmus konstruieren, der feststellt, ob in einem Graphen ein *perfektes Matching* existiert oder nicht. Ob dieser Algorithmus gegenüber deterministischen Algorithmen Laufzeitvorteile hat hängt davon ab, wieviele Kanten ein Graph einthält. Ein deterministischer Algorithmus – einer der schnellsten bisher konstruierten läuft mit $O(m\sqrt{n})$ – wird meistens etwas mit "Durchprobieren" verschiedener

Kantenkombinationen zu tun haben. Wenn es viele Kanten sind, ist es sicher besser die Determinanten für einen bestimmten Wert zu berechnen. Wenn es wenige Kanten sind, kann die Determinantenbestimmung leicht langsamer sein als eine deterministische „Suche“.

3.3 Fingerprintingverfahren auf Basis der Verkleinerung des zugrundeliegenden Körpers

Bisher wurden Fingerprints immer dadurch erzeugt, dass aus dem der Anwendung zugrundeliegenden Körper ein zufälliges Element ausgewählt wird, das dann mit algebraischen Mitteln die Komplexität einer Anwendung mit einer bestimmten Fehlerwahrscheinlichkeit reduziert.
Im Verfahren um das es jetzt gehen wird, wird dagegen der zugrunde liegende Körper zufällig variiert (verkleinert), wodurch man beispielsweise für den Vergleich von Strings und Patternmatching interessante probabilistische Algorithmen erhält.

3.3.1 Gleichheit von Strings

Für einen Algorithmus zur Überprüfung der Gleichheit von Strings gibt es zahlreiche Anwendungsfälle. Ein Beispiel ist die Replikation von Datenbanken[4]. Dabei geht es darum, zwei Datenbanken zu vergleichen und sie bei festgestellter Inkonsistenz zu synchronisieren. Die Datenbanken sind nicht selten räumlich voneinander getrennt und verfügen nicht unbedingt über eine allzu schnelle und breite Verbindung. Einfach nur einen Teil der Daten zu übertragen birgt ein hohes Fehlerrisiko, wenn zufällig genau die Teile übertragen werden, die keinen Unterschied enthalten. Es gibt aber auch ein probabilistisches Verfahren, das Inkonsistenzen mit sehr hoher Wahrscheinlichkeit erkennt, wobei nur relativ kleine ”Fingerabdrücke” übertragen werden müssen.

Algorithmus 3 (Stringvergleich) *Seien* $a = (a_1, \ldots, a_n)$ *und* $b = (b_1, \ldots, b_n)$ *die zu vergleichenden Strings als Bitfolge der Länge* n.
Seien $A = \sum_{i=1}^{n} a_i 2^{i-1}$ *und* $B = \sum_{i=1}^{n} b_i 2^{i-1}$ *die Strings interpretiert als ganze Zahl.*
Sei $2 \leq p \leq \tau$ *eine zufällig ausgewählte Primzahl.*
Sei $F_p(x) = x \mod p$ *eine Fingerprintingfunktion.*

if $F_p(A) = F_p(B)$ *return true*
else return false

□

[4]Eine Datenbank wird hier vereinfacht als ein langer zusammenhängender String betrachtet.

Der Fingerabdruck eines Strings ist also die Zahl, die man erhält, wenn man den String als ganze Zahl interpretiert und modulo einer Primzahl rechnet. Diese Fingerprints haben als Bitfolge dargestellt eine Länge von $\log p$, was je nach Wahl von p viel kleiner als n ist.
Auch hier gilt wieder, dass für $A = B$ immer true zurückgeliefert wird. Ein Fehler kann nur auftreten, wenn $A \neq B$ aber dennoch $F_p(A) = F_p(B)$ ist. Der nun folgende Satz macht eine Aussage über diese Fehlerwahrscheinlichkeit in Abhängigkeit vom Bereich aus dem die verwendete Primzahl zufällig ausgewählt wurde.

Satz 5 (Stringvergleich) *Sei $F_p(x) = x \mod p$ wobei p eine Primzahl ist, die zufällig aus der Menge der Primzahlen zwischen 0 und τ ausgewählt wurde. Sei τ so gewählt, dass $\tau = tn \log tn$ für ein beliebiges großes t. Sei weiterhin $\pi(x)$ eine Funktion, die die Anzahl der Primzahlen liefert, die kleiner als x sind. Dann gilt:*

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

Beweis: Betrachte $C := |A - B|$. Der obige Fingerprintingalgorithmus liefert ein falsches Ergebnis, wenn die zufällig ausgewählte Primzahl C teilt und $C \neq 0$.
C hat wie jede Zahl $< 2^n$ weniger als n Primfaktoren. Die Anzahl der möglichen Primzahlen ist $\pi(\tau)$. Von diesen $\pi(\tau)$ führen also höchstens n zu einem Fehler. Es gilt also:

$$Pr[F_p(A) = F_p(B) | a \neq b] \leq \frac{n}{\pi(\tau)}$$

Nach dem Primzahlsatz gilt $\pi(x) \sim \frac{x}{\ln x}$.

$$\Rightarrow \frac{n}{\pi(\tau)} \approx \frac{n \ln \tau}{\tau} = \frac{n \ln (tn \log tn)}{tn \log tn} = O(\frac{1}{t})$$

□

Zuguterletzt sollte man noch etwas dazu sagen, warum *mod Primzahl* und nicht *mod irgendeiner Zahl* gerechnet wird.
Ein erster Grund, der im Zusammenhang mit der Konstruktion von Hashfunktionen oft behandelt wird ist, dass man die Urbilder von F_p im Bild möglichst „gleichverteilt“ vorfinden will und dass alle Bits der BitDarstellung des Strings „gleichen Einfluß“ auf das Ergebnis von F_p haben. Das funktioniert am besten mit einer Primzahl, die möglichst weit von einer Zweierpotenz entfernt liegt.
Ein zweiter Grund ist die Tatsache, dass $\mathbb{Z}/(p)$ ein Körper ist, wenn p eine Primzahl ist. Bei dieser Anwendung spielt das vielleicht keine Rolle. Es wäre aber beispielsweise denkbar die Methode der *Variation des Grundkörpers* mit einer anderen Fingerprintingmethode zu kombinieren. Dies könnte geschehen, indem man bei einem algebraischen Berechnungsvorgang, wie der

Berechnung der Determinante, nach jedem Rechenschritt $\mod p$ rechnet. So kann man übrigens auch das Entstehen sehr großer Zwischenergebnisse vermeiden. Bei solchen Anwendungen kann es aber Probleme machen, wenn die Grundmenge kein Körper mehr ist, also wenn z.B. Nullteiler ein Berechnungsergebnis unbrauchbar machen könnten.

3.3.2 Patternmatching

Beim Patternmatching handelt es sich um die Suche in einem Text nach dem Vorkommen eines Patterns. Das heisst, dass in einem als String $X = (x_1, \ldots, x_n)$ dargestellten Text der Länge n nach dem Vorkommen des als String der Länge m gegebenen Patterns $Y = (y_1, \ldots, y_m)$ gesucht wird, wobei natürlich $m < n$. Ein *Match* tritt auf, wenn ein $j \in \{1, 2, \ldots, n-m+1\}$ existiert bzw. gefunden wird, so dass $x_{j+i-1} = y_i$ für $1 \leq i \leq m$. Der „naive“ Patternmatching-Algorithmus, der einfach an allen m Stellen des Textes überprüft, ob das Pattern matcht oder nicht hat eine Laufzeit von $O(mn)$. Es gibt auch schnellere deterministische Algorithmen mit einer Laufzeit von $O(m+n)$.
Der im folgenden beschriebene probabilistische Algorithmus wird auch eine Laufzeit von $O(m+n)$ haben. Er ist aber erheblich simpler als die deterministischen Algorithmen mit gleicher Laufzeit, und er bietet in einigen Anwendungsfällen deutliche Vorteile. O.B.d.A enthalten Strings hier nur Symbole aus dem Alphabet $\sum = \{0, 1\}$

Algorithmus 4 (Patternmatching) *Sei X ein String der Länge n und Y ein Pattern der Länge m mit $m < n$.*
Sei $X(j) = x_j x_{j+1} \ldots x_{j+m-1}$ ein Teilstring von X der Länge m.
Sei $F_p(Z) = Z \mod p$ eine Fingerprintingfunktion, wobei p eine Primzahl ist, die aus den Primzahlen zwischen 0 und τ zufällig ausgewählt wurde. Z ist in diesem Fall ein String, der als m-Bit Integerwert interpretiert wird.

for($j = 0; j \leq n - m; j++$)
* if $F_p(X(j)) = F_p(Y)$ then return true*
return false

□

Satz 6 (Patternmatching) *Der Patternmatching-Algorithmus hat eine Laufzeit von $O(n+m)$ und eine Fehlerwahrscheinlichkeit von $O(\frac{1}{n})$.*

Beweis: 1.) Fehlerwahrscheinlichkeit: Dieser Algorithmus liefert beim ersten korrekten Match immer true zurück. Er schlägt nur fehl, wenn $F_p(X(j) = F_p(Y)$ obwohl $X(j) \neq Y$. Nach Satz 5 gilt für die Fehlerwahrscheinlichkeit:

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

Die Wahrscheinlichkeit, dass bei einem der $1 \leq j \leq n$ Vergleichsvorgänge ein falscher Treffer auftritt, ist dann $O(\frac{nm \log \tau}{\tau})$. Setzt man $\tau = n^2 m \log n^2 m$, erhält man

$$Pr[\text{Falscher Treffer}] = O(\tfrac{1}{n}).$$

2.) Laufzeit: Bevor man die Laufzeit des obigen Algorithmus analysieren kann, sollte man die enthaltenen Berechnungsvorgänge etwas genauer spezifizieren. Der Algorithmus lässt sich sehr effizient gestalten, indem man $X(j)$ mittels einer Rekursionsformel aus $X(j-1)$ berechnet:

$$X(j) = 2[X(j-1) - 2^{m-1} x_{j-1}] + x_{(j-1)+m}$$

Damit ergibt sich für die Berechnung von $F_p(j)$ folgende Rekursionsformel:

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

Damit ist $X(j+1)$ bei gegebenem $X(j)$ mit $O(1)$ berechenbar. Der Algorithmus hat also eine Laufzeit von $O(n+m)$.

□

Mit diesen Ergebnissen kann man nun verschiedene Algorithmen konstruieren. Der einfachste Algorithmus ist der oben beschriebene Monte Carlo Algorithmus. Man kann diesen zu einem Las Vegas Algorithmus erweitern: Man startet mit dem Monte Carlo-Algorithmus. Bei einem gefundenen Treffer überprüft man, ob gilt $X(j) = Y$. Im Fehlerfall gibt es verschiedene Möglichkeiten. Zum einen kann man im Fehlerfall einen deterministischen Algorithmus aufrufen, was sich auf die Worst-Case-Laufzeit aber nicht sonderlich positiv auswirkt. Eine andere Möglichkeit besteht darin, den Algorithmus mit einer anderen Wahl von p neu aufzurufen. Man kann Algorithmus 4 auch zu einer deterministischen Variante umbauen. Man überprüft bei jedem Treffer, ob es sich auch wirklich um einen Treffer handelt. Dieser Algorithmus hat eine Worst-Cast-Laufzeit $O(mn)$. Man kann aber bei einer guten Auswahl von $F_p(X)$ die Laufzeit erheblich in Richtung $O(n+m)$ reduzieren. Letzterer Algorithmus ist auch als Rabin-Karp-Algorithmus bekannt.

Literatur

[1] R. Motwani, P. Raghavan: *Randomized Algorithms*, Camebridge University Press, 1995, ISBN 0-521-47465-5

[2] T.H. Cormen, C.E. Leisserson, R.L. Rivest, C. Stein: *Introduction to Algorithms*, MIT-Press, 2001, ISBN 0-07-013151-1

[3] *Brockhaus-Kalender — Was so nicht im Lexikon steht 2004*, ISBN 3-7653-3304-2

[4] *Meyers neues Lexikon — Band 3: Fe-Hn*, Meyers Lexikonverlag, 1979, ISBN 3-411-01753-8

[5] N. Henze: *Stochastik für Einsteiger*, Vieweg Verlag, 2003, ISBN 3-528-36894-2