Aufgabe: CRC

Fehlererkennende Codes: Idee

Im Detail werden Codierungen, insbesondere lineare Block-Codes, im Rahmen von MafI-2 „Lineare Algebra“ erläutert. Hier wird nur ein grober, unmathematischer Abriss geschildert, der nur zum Verständnis gereichen soll, warum wir überhaupt CRCs berechnen möchten.

Unser Problem ist, dass bei Datenübertragung häufig Fehler auftauchen können, diese treten in Form von Bitflips auf, d. h. die Bits kippen. In der Regel passiert das nicht mit einzelnen Bits, sondern eine ganze Reihe an Bits kippen, und dies auch nicht verteilt über die Nachricht, sondern in „Bursts“ auf einem Fleck.

Das Ziel ist, zusätzlich zu der Nachricht eine Check-Zahl, die sich mit einem festen Algorithmus aus der Nachricht berechnen lässt, zu versenden. Der Empfänger kann nun den gleichen Algorithmus nutzen und die Ergebnisse vergleichen und somit (hoffentlich) erkennen, ob die Nachricht korrekt übertragen wurde oder nicht.

Die Lösung die fehlererkenndende Codes bieten, ist dass wir der Nachricht (welche unendlichlang sein kann!) über eine Codierungsfunktion eine endliche Zahl zuordnen. Offensichtlich ist es (Taubenschlagprinzip!) nicht möglich, dass jede Nachricht ihren eigenen Code bekommt, d. h. es gibt Nachrichten, die auf die gleiche Zahl abgebildet werden müssen. Das besondere an CRCs ist, dass die Wahrscheinlichkeit, dass nur durch die o. g. Bursts praktisch keine „Kollisionen“ auftreten können, d. h. eben genau solche Nachrichten, die auf die gleiche CRC abbilden, wie das Original.

CRC

Polynomdivision

CRCs sind im Prinzip nichts weiter als Polynomdivision, wobei wir die Eingangsnachricht aus dem Alphabet $\Sigma = \{0, 1\}$ als Koeffizienten für ein Polynom auffassen, d. h. wir die Nachricht „10101“ wird als Polynom $M(x) = 1 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0$ interpretieren. Zudem haben wir ein generierendes Polynom $G(x)$ von einem Grad n, welches die grundlegenste Spezifikation des konkreten CRCs darstellt (CRC-16 haben alle einen Polynomgrad von 16).

Der Algorithmus basiert auf der Gleichung des Euklidischen Algorithmus' für Reste bei Polynomdivision:

$$M(x) \cdot x^n = Q(x) \cdot G(x) + R(x)$$

Das Nachrichtenpolynom $M(x)$ multipliziert mit x^n ergibt ein Polynom vom Grad $\deg(M) + n$. Dieses Polynom kann durch Division mit dem Generatorpolynom $G(x)$ aufgespalten werden in $Q(x)$ und den Rest $R(nx)$; letzterer ist das, was uns eigentlich interessiert, somit schreiben wir stattdessen:

$$R(x) = M(x) \cdot x^n \bmod G(x)$$

Dieser Rest ist das Ergebnis unseres Algorithmus' und wird zusätzlich zu unserer Nachricht mitgeschickt. Der Empfänger kann nun mit $G(x)$ und $M(x)$ die gleiche Berechnung anstellen und die Ergebnisse vergleichen. Alternativ wissen wir, dass natürlich $M(x) \cdot x^n - R(x) = Q(n) \cdot G(n)$ sein muss, das heißt die $M(x) \cdot x^n - R(x)$ bei Division durch das Generatorpolynom $G(n)$ schlicht 0 ergeben muss. In Binärdarstellung, entspricht der Term $M(x) \cdot x^n - R(x)$ schlicht der Konkatenation der Bitstrings der Nachricht und der CRC.

Variationen

Führende Nullen: Invertierung des Startwertes Wie wir alle wissen, sind führende Nullen bei einer Berechnung egal, sprich es ist aktuell noch sehr einfach eine Nachricht zu erstellen, die das gleiche Codewort ergibt, solange wir unterschiedliche Nachrichtenlänge in Kauf nehmen: Man braucht nur Nullen vorne anzufügen. Um dies zu Umgehen, kann man schlicht die ersten $n = \deg G(n)$ Bits der Eingangsnachricht invertieren; solange Empfänger und Sender dies wissen, funktioniert der Algorithmus sonst wie gehabt. Dies ist sehr einfach zu implementieren, indem man Schlicht einen anderen Anfangswert setzt für das sog. Shift-Register setzt, aber dazu kommen wir später.

Nachgestellte Nullen: Invertierung des Ergebnisses Aus ähnlichen Gründen gibt es Probleme bei Nullen die an die Nachricht angehängt werden, invertiert (d. h. kippt alle Bits) man einfach den CRC-Wert am Ende, ist das kein Problem mehr.

Notation & Hardware

Es gibt mehrere Notationen für die Polynome. Da es immer einen Term von x^0 und einen Term von x^n gibt, kann man diese eigentlich weglassen. Es wird aber in jeder Notation nur einer der Koeffizienten implizit angenommen.

Außerdem werden je nach Hardware, die Bits in einem Byte uintuitiv sortiert, und eben nicht mit dem Most-Significant-Bit zuerst, sondern andersherum (LSB first). Analog werden deshalb auch die Polynome manchmal umgekehrt, und entsprechend wieder das gesetzte LSB oder MSB implizit angenommen.

Es gibt also von einem Polynom 4 Kombinationen der Schreibweisen:

	MSB impl.	LSB impl.
MSB first	normal	reciprocal
LSB first	reverse	rev.-reciprocal

Man beachte dabei, dass sich bei den reziproken Polynomen das signfikante Bit plötzlich sich auch umdreht.

Für das Polynom $x^{16} + x^{15} + x^2 + 1$ ergibt sich somit in Bitdarstellung:

```
 1 [1  0  0  0| 0  0  0  0| 0  0  0  0| 0  1  0  1]      normal
   [    8     |     0     |     0     |     5     ]
[1  0  1  0| 0  0  0  0| 0  0  0  0| 0  0  0  1] 1      reverse
[    A     |     0     |     0     |     A     ]
 1 [0  1  0  0| 0  0  0  0| 0  0  0  0| 0  0  1  1]      reciprocal
   [    4     |     0     |     0     |     3     ]
[1  1  0  0| 0  0  0  0| 0  0  0  0| 0  0  1  0] 1      rev.-reciprocal
[    C     |     0     |     0     |     2     ]
```

Implementierung: Ansatz

Je nach Optimierungsziel gibt es neben der naiven, rein mathematischen Variante auch Möglichkeiten anders, den Algorithmus auszuführen, dies soll aber hier nicht genauer behandelt werden.

In aller Regel wird in der Hardware ein sog. Shift-Register verwendet, welches den aktuellen Rest vorhält. Dieser wird benutzt um den neuen Rest zu berechnen, während man in der Nachricht durch Bit-Operationen ein Bit weiter wandert.

Da nun häufig das MSB (oder auch LSB) nicht explizit angegeben ist, muss dies ggf. auch in dem Algorithmus berücksichtigt werden.

Wie schon vorher angedeutet gibt es eine sehr einfache Art, das Problem der führenden Nullen zu vermeiden: Die dort angedeutete Lösung entspricht in Programmcode schlicht die Initialisierung des Shift-Registers bzw. der entsprechenden Variable auf einen Wert verschieden von Null (häufig das Komplement von Null).

Genauso kann um das Problem der nachgestellten Nullen behoben werden in dem am Ende einmal der CRC mit einer festgelegten Zahl ge-xor-t werden (ebenfalls häufig das Komplement von Null).

CRC16-IBM: Spezifikation

CRC16-IBM nutzt das Polynom `0x8005` mit dem Startwert `0xFFFF` und dem End-XOR-Wert `0xFFFF`. Zudem wird jedes Eingabebyte in LSB-first bearbeitet, sprich entweder man arbeitet mit MSB-first und kehrt bei jeder Division die Bits um, oder man arbeitet mit dem Polynom `0xA001` und shiftet innerhalb der Bytes in die andere Richtung (genaures ist dann eure Arbeit!)

Weitere Informationen

- `https://en.wikipedia.org/wiki/Cyclic_redundancy_check`
- `https://en.wikipedia.org/wiki/Computation_of_cyclic_redundancy_checks`
- `https://en.wikipedia.org/wiki/Cyclic_code`
- `https://en.wikipedia.org/wiki/Mathematics_of_cyclic_redundancy_checks`