Departamento de Arquitectura Instituto de Computación

Universidad de la República

Montevideo - Uruguay

Notas de Teórico

Códigos y Errores

Arquitectura de Computadoras

(Versión 4.3b - 2016)

2 CODIGOS Y ERRORES

2.1 Introducción

En el próximo capítulo estudiaremos la forma en que se representan los distintos tipos de objetos con que trabajamos en computación. En los lenguajes de alto nivel manejamos distintos tipos de datos: caracteres, strings, números enteros, números reales. Para trabajar con ellos, en general sólo nos interesa saber que son, como se opera con cada uno y esto se estudia en los cursos de programación. En esta materia nos ocuparemos de otro aspecto: como están implementados, a nivel interno, los distintos tipos de datos.

La unidad elemental de información que se usa en computación es un objeto que toma solo 2 valores posibles (0, 1): el BIT (dígito binario)

Los distintos tipos de datos se construyen en base a estructuras de bits, los cuales serán en general “tiras” (sucesión ordenada de bits) de n elementos que reciben el nombre de **palabra** de largo n. En el caso particular de n = 8 la tira se denomina **byte**.

Estudiaremos entonces, la representación interna de los datos como la expresión de los distintos tipos en función de estructuras de bits.

Por lo anterior resulta que los distintos tipos de datos se representan a través de **códigos binarios**. Es decir existe un proceso de codificación de los objetos de información en función de otros (las estructuras de bits) con los que se trabajará en realidad.

Conviene alertar que el término “código” se usa indistintamente para referirse tanto a un código binario particular (la respresentación de un objeto), como a un sistema de codificación. Que sea una u otra acepción dependerá del contexto.

Por esta razón es interesante estudiar, aunque sea brevemente, un problema fundamental en la manipulación de códigos binarios: la detección y corrección de errores.

No nos referiremos aquí a errores en la codificación de los objetos sino a los que aparecen cuando se manipula con ellos. Usualmente los objetos de información se almacenan y/o se transmiten. Estas dos operaciones comunes se realizan, en definitiva, utilizando medios físicos (memorias, discos, canales de comunicación, etc.) los cuales no están libres de errores. Por tanto es de particular interés el estudio de la posibilidad de detectar o corregir errores en códigos binarios. De esta manera nos aseguraríamos que un dato recuperado de una unidad de almacenamiento es correcto (coincide con el almacenado) o que un dato recibido por un canal de comunicaciones lo es (coincide con el enviado por el emisor).

2.2 Detección y Corrección de Errores

Todos los sistemas de codificación que permiten detección y corrección de errores se basan en una misma idea: redundancia.

El fundamento es sencillo: para poder distinguir si un valor es correcto o no (o sea representa un objeto válido del sistema o no) debo agregar información adicional al código. Entonces en todo sistema de codificación con capacidad de detectar errores el número de objetos representados es siempre menor que el número de valores posibles del código binario utilizado.

Esta afirmación quedará más clara cuando veamos las distintas estrategias utilizadas. De todas formas un ejemplo puede ser de utilidad: supongamos que tenemos 16 objetos a representar; en principio, con un código binario de 4 bits alcanzaría ($2^4 = 16$), pero con 4 bits no estaríamos en condiciones de detectar errores puesto que todos los valores posibles del código binario utilizado corresponderían a objetos válidos. De esta manera si, por ejemplo, hubiera un error en un bit del código que representa a un objeto A se transformaría en el código que representa a un objeto B por lo que no habría posibilidad de detectarlo.

Para disponer, en este caso de la posibilidad de distinguir un código binario correcto de uno incorrecto, deberíamos utilizar un código binario de, por ejemplo, 6 bits. Con 6 bits tendríamos 64 valores posibles de los cuales sólo 16 representarían objetos reales. Si elegimos convenientemente los códigos asociados a cada objeto estamos en condiciones de detectar errores producidos por el cambio de un bit en el código que representa un objeto.

2.3 Distancia

La distancia entre dos representaciones binarias se define como el número de bits distintos entre los dos códigos.

Es decir si tenemos dos códigos binarios a y b

$$(a_0 , a_1, \ldots, a_n)$$

$$(b_0, b_1, \ldots, b_n)$$

la distancia entre ellos está dada por la cantidad de unos en el código formado por:

$$(a_0 \oplus b_0 , a_1 \oplus b_1 , \ldots a_n \oplus b_n)$$

donde $\oplus$ representa a la operación *O Exclusivo* (XOR).

Por ejemplo, los códigos:

01101
10110

tienen una distancia 4 (cuatro)

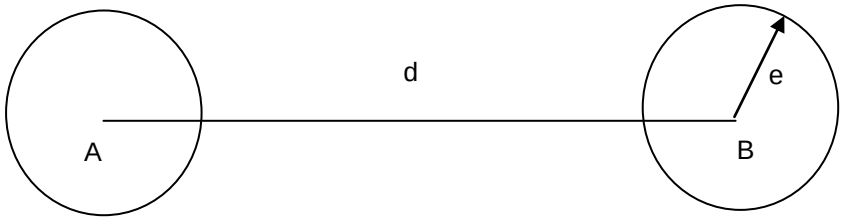
La "distancia" tal cual la hemos definido tiene las siguientes propiedades (que no vamos a demostrar):

1) $d(a,b) = d(b,a)$
2) $d(a,b) = 0$ *si y sólo si* $a = b$
3) $d(a,b) + d(b,c) \geq d(a,c)$

Se pueden generar sistemas de codificación binarios que tengan una determinada distancia. En el caso de un sistema de codificación se llama "distancia" del código (ó del sistema de codificación) a la mínima distancia que exista entre dos valores válidos del código.

La posibilidad de detectar y corregir errores está fuertemente vinculada, como es de suponer, a la distancia del código (sistema de codificación) utilizado.

Consideremos un código (sistema de codificación) de distancia "d" y dos objetos válidos del código A y B. Consideremos el conjunto de valores posibles del sistema de codificación (no necesariamente válidos) que se obtienen de modificar hasta "e" bits del valor "A". Idem para los del valor "B". Estos conjuntos se pueden visualizar como "esferas" en el espacio de "*n*" dimensiones (siendo n la cantidad de bits del código) rodeando a cada uno de los valores.



Como se puede ver, si estos conjuntos no tienen puntos en común, podemos no solo detectar la existencia de un error, sino que además podemos reconocer a qué código correcto corresponde un valor "degenerado" dado, permitiendo por tanto la "corrección" del error. Si los conjuntos tienen puntos en común no podremos corregir, pero si detectar errores, siempre y cuando dentro del conjunto de puntos asociado al valor A no se incluya al valor B, puesto que si no, la alteración de un código válido puede conducir a otro válido, impidiendo así detectar el error.

Es decir que la condición para que se puedan detectar errores es que:

$$e < d$$

la condición para que se puedan corregir es:

$$e < \frac{d}{2}$$

Por ejemplo para poder corregir errores de hasta un bit, hay que utilizar un código de distancia 3 por lo menos.

2.4 Bits de Redundancia

Se puede demostrar que para generar códigos de distancia 3 para objetos representables en "k" bits, se necesitan utilizar "p" bits adicionales, llamados bits de redundancia ó de paridad, tal que se cumpla:

$$2^p \geq p + k + 1$$

El razonamiento detrás de esta afirmación es que para poder identificar el bit que tiene error dipondremos de 2^p combinaciones posibles, generadas por los *p* bits. Con esas combinaciones debemos poder señalar cual es el bit errado de los $p + k$ bits que tendrá el código completo (el original más los bits de redundancia agregados). Además precisaremos tener una combinación para indicar que no hay error y de ahí surge que el lado derecho de la expresión es $p + k + 1$.

Por ejemplo, supongamos que queremos diseñar un sistema de memoria de una computadora de 16 bits, de manera que sea capaz de corregir errores de hasta 1 bit. De acuerdo a la expresión necesitamos agregar 5 bits de redundancia para que esto sea posible.

$$2^5 = 32 \geq 5 + 16 + 1 = 22$$

La siguiente tabla muestra los valores de p (bits agregados) para distintos valores de k (bits del código original).

k	p
4	3
8	4
16	5
32	6
64	7
128	8

En la tabla se observa que la cantidad relativa de bits que es necesario agregar disminuye a medida que aumenta el tamaño del código original. Esta propiedad tiene impacto sobre el hecho que en las computadoras de hoy en día, que son de 32 ó 64 bits de tamaño de palabra, sea más frecuente encontrar sistemas de memoria *ECC* (*Error Correcting Code*) que permiten la detección y corrección de un bit fallado.

2.5 Paridad

Uno de los mecanismos para generar sistemas de codificación con capacidad de detectar errores es el de la paridad, o codificación con bit de paridad.

La idea es agregar a cada código binario de n bits que representan objetos válidos, un nuevo bit calculado en función de los restantes. La forma en que se calcula el bit de redundancia (de paridad) es tal que la cantidad de unos en el código completo (original + bit de paridad) sea par (en cuyo caso hablamos de **paridad par**) o impar (en cuyo caso se trata de **paridad impar**).

Cuando se genera el código (se almacena o se transmite) se calcula este bit mediante uno de los dos criterios expuestos a partir de los n bits originales. Cuando se recupera el código (se lee o se recibe) se recalcula el bit y se chequea con el almacenado o transmitido. En caso de no coincidir estamos ante un error, si el chequeo cierra podemos decir que para nuestro sistema no hubo errores.

Notemos que no podemos afirmar en forma absoluta la ausencia de error aunque el bit de paridad recalculado coincida con el recuperado. Esto es una regla general para todo sistema de codificación con detección de errores, y en este caso se ve claramente: si cambian a la vez 2 bits del código o, en general, un número par de ellos, el sistema de detección de errores por paridad no funciona, en el sentido que no reporta el error.

Los sistemas de codificación con capacidad de detección (o detección y corrección) de errores funcionan correctamente dentro de ciertos límites, si se cumplen ciertas hipótesis de trabajo. Un conjunto de supuestos que de cumplirse aseguran el buen funcionamiento de los sistemas que estamos analizando es:

(a) La probabilidad de que falle un bit es baja.
(b) Las fallas de bits son sucesos independientes (la posibilidad que un bit falle no tiene relación alguna con la falla de otro bit del código).

En esta hipótesis la probabilidad de que fallen 2 bits a la vez es igual al cuadrado de la probabilidad que falle uno (por la segunda hipótesis) que ya era un valor pequeño (por la primera) por lo que da un valor muy bajo. Por lo tanto en estas hipótesis el sistema de detección por paridad funciona bien: cuando no detecta error es altamente probable que el código sea efectivamente el correcto.

Es importante señalar que las hipótesis efectuadas se ajustan al caso de las memorias de las computadoras modernas. No ocurre lo mismo con los dispositivos de almacenamiento que graban la información en forma serial (un bit a continuación del otro) ni con los sistemas de transmisión de datos seriales ya que en estos casos el hecho que falle un bit está vinculado, en forma no despreciable, a la falla de otro (en particular del vecino "anterior"), por lo cual la hipótesis b) no se cumple para estos sistemas, de donde el control por paridad no sería demasiado efectivo en estos casos.

Volviendo al mecanismo de bit de paridad y recordando la definición de la operación lógica XOR y su propiedad asociativa es fácil ver que el bit de paridad se puede expresar como:

$P = b_0 \oplus b_1 \oplus \ldots \oplus b_n$ *(paridad par)*

ó

$P = (b_0 \oplus b_1 \oplus \ldots \oplus b_n)'$ *(paridad impar)*

El chequeo de la corrección de un código recuperado se puede realizar evaluando la expresión (para paridad par):

$P \oplus b_0 \oplus b_1 \oplus \ldots \oplus b_n$

con los valores de P, $b_0 \ldots b_n$ recuperados. Si el resultado es 0 "no hubo error", si es 1 entonces se detectó un error.

Una variante de este sistema es el mecanismo de paridad horizontal/vertical. Este método se puede aplicar cuando se almacenan o transmiten varias palabras de código. Cada palabra de n bits de código tiene su bit de paridad (que se denomina paridad horizontal) y cada n palabras almacenadas/transmitidas se agrega una palabra de paridad generada como el XOR (bit a bit, incluyendo el de paridad horizontal) que recibe el nombre de paridad vertical. Por ejemplo para n=8 sería algo así:

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_h
b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_h
				.....				
				.....				
v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_h

$a_h = a_0 \oplus a_1 \oplus . . . \oplus a_7$ (idem para b, c, d, e, f, g, h)

$v_i = a_i \oplus b_i \oplus . . . \oplus h_i$ (i= 0, 1, 2, 3, 4, 5, 6, 7, h)

2.6 2 de 5

Los sistemas de codificación con capacidad de manejar los errores pueden ser construidos a partir de un sistema de codificación que no tenga esa capacidad (como es el caso de la "paridad" ya visto) o pueden contruirse desde el comienzo con una distancia mayor o igual a 2 de forma que tengan incoporada la caractaerística desde el comienzo.

Un ejemplo de esta segunda categoría es el código "*2 de 5*", el cuál puede tener la forma:

01100 - 0
11000 - 1
10100 - 2
10010 - 3
10001 - 4
01010 - 5
01001 - 6
00110 - 7
00101 - 8
00011 - 9

Como vemos este código tiene distancia 2 ya que todos los elementos del código difieren por lo menos en 2 bits (ej: *11000* y *10100*). De los 32 posibles objetos que permiten representar 5 bits, este código permite representar tan solo 10 objetos.

Una aplicación práctica de un sistema de codificación es el "*Entrelazado 2 de 5*" ("*Interleaved 2 of 5*") que es un código de barras utilizado para identificar artículos. La codificación que se utiliza es:

0	ffAAf
1	AfffA
2	fAffA
3	AAfff
4	ffAfA
5	AfAff
6	fAAff
7	fffAA
8	AffAf
9	fAfAf

donde "f" significa una barra fina y "A" una ancha.

El "entrelazado" del nombre se refiere a que la codificación se realiza de a parejas de dígitos, combinando las barras oscuras con las barras claras. Al comienzo del código se alternan barras finas oscuras y claras para indicar el comienzo.

Notar que luego de la secuencia de inicio fina oscura, fina clara, fina oscura, fina clara, vienen las siguientes barras oscuras: AfffA, intercaladas con las siguientes barras claras: fAffA. Esto es porque se toman los dos primeros dígitos (12) y se representa al 1 con las barras oscuras y al 2 con las claras intercalando oscuras con claras.

2.7 Códigos de Hamming

Los códigos de Hamming son una forma práctica de generar códigos de distancia 3. Los veremos a través de un ejemplo en concreto. Supongamos que tenemos 16 objetos representados, en principio, en binario:

$$a_4\, a_3\, a_2\, a_1$$

de acuerdo a lo visto antes precisamos 3 bits de redundancia (ya que $2^3 = 8 \geq 3 + 4 + 1 = 8$)

$$p_3\, p_2\, p_1$$

Si armamos el código (que tendrá entonces 7 bits) de la siguiente manera:

$$a_4\, a_3\, a_2\, p_3\, a_1\, p_2\, p_1$$

y convenimos que las representaciones válidas de los 16 objetos a representar son aquellas en los cuales los bits de redundancia se calculan de la siguiente manera:

$$p_1 = a_4 \oplus a_2 \oplus a_1$$

$$p_2 = a_4 \oplus a_3 \oplus a_1$$

$$p_3 = a_4 \oplus a_3 \oplus a_2$$

podemos al recuperar el código calcular los bits "s" que están vinculados, al igual que los "p" a la posición de los distintos bits en el código:

	a_4	a_3	a_2	p_3	a_1	p_2	p_1
S_0	x		x		x		x
S_1	x	x			x	x	
S_2	x	x	x	x			

$$s_0 = p_1 \oplus a_4 \oplus a_2 \oplus a_1$$

$$s_1 = p_2 \oplus a_4 \oplus a_3 \oplus a_1$$

$$s_2 = p_3 \oplus a_4 \oplus a_3 \oplus a_2$$

estos bits "s" representan, en binario, un número "S"

$$S = S_2 S_1 S_0$$

Si al calcular "S" se da que es cero no hay errores (por lo menos para nuestro sistema de codificación). Si "S" da distinto de 0, el número que de me indica el bit que está errado (siendo el 1 el de más a la derecha, o sea p_1). De esta manera es posible reconstruir el valor correcto del código, cambiando el bit que identificamos como corrupto (si está en 0 lo pasamos a 1 y viceversa).

La propiedades de este sistema de codificación están determinadas por la elección de las posiciones de los bits del código original y los bits de paridad en el código resultante. Como se puede observar la tabla que utilizamos permite reconocer la posición de cada dígito si reemplazamos las x por 1 y leemos el número $S_2S_1S_0$ en binario.

2.8 Código de Verificación de Suma

Como fue mencionado los sistemas de detección por paridad son aptos para trabajar en la hipótesis que la falla de un bit es un suceso probabilísticamente independiente de la falla de un bit "vecino". Los códigos de Hamming también se apoyan en esta hipótesis para asegurar eficacia en la detección y corrección del error de un bit.

Esta hipótesis se cumple muy bien en dispositivos como ser las memorias de las computadoras (RAM). Sin embargo en los sistemas de almacenamiento masivo en base a discos magnéticos u ópticos, o en sistemas de transmisión serie, la hipótesis no es cierta y por tanto estos mecanismos de manejo de errores no son del todo efectivos.

Para estos casos se utilizan otros métodos, diseñados para ser capaces de manejar errores múltiples ó "errores en ráfaga" (un error en ráfaga significa que fallan varios bits contiguos en forma simultánea).

El más sencillo de estos métodos es el denominado Código de Verificación de Suma, también conocido por su denominación en inglés “checksum”. El “checksum” consiste en agregarle a un conjunto de códigos binarios almacenados ó transmitidos un código adicional consistente en la suma binaria de los mismos, módulo 2^n (siendo n la cantidad de bits del código).

Supongamos un mensaje (Hola!!) formado por 6 caracteres representados en ASCII de 8 bits. El código binario quedaría:

01001000 01101111 01101100 01100001 00100001 00100001

y el “checksum” a agregar al final se calcula como:

```
  01001000
+ 01101111
+ 01101100
+ 01100001
+ 00100001
+ 00100001
-----------
```

111000110 que si nos quedamos con los 8 bits menos significativos es: 11000110

Por lo que el mensaje incluyendo el checksum sería:
01001000 01101111 01101100 01100001 00100001 0010000111000110

Si bien es simple de implementar no es muy efectivo (es fácil ver que hay combinaciones de bits erróneos que computan el mismo “checksum”).

2.9 Código de Redundancia Cíclica

Un código para errores en ráfaga muy difundido es el Código de Redundancia Cíclica (CRC), que se basa en la aritmética de módulo aplicada a los polinomios.

Consideremos un "mensaje" a transmitir (o almacenar) de **m** bits. Llamamos M(x) al polinomio de grado m-1 cuyos coeficientes coinciden con los bits del mensaje. Se define un polinomio G(x) de grado **r** de coeficientes binarios (0 ó 1), que es característico del sistema.

Consideremos el polinomio $x^rM(x)$, en otras palabras elevamos el grado del polinomio M(X) en r. Planteemos la división entera de dicho polinomio entre G(x):

$x^rM(x) = Q(x)G(x) + R(x)$ donde Q(x) = cociente y R(x) = resto

Consideremos ahora el polinomio T(x) definido por la expresión:

$T(x) = x^rM(x) - R(x)$

Es sencillo comprobar que T(x) es divisible por G(x). T(x) es el polinomio que se transmite (o almacena). En realidad los que se transmiten (o almacenan) son los bits coeficientes de T(x). Notemos que los primeros m bits de T(x) son los mismos que M(x), ya que R(x) es a lo sumo de grado r-1 y por tanto no afecta ninguno de los coeficientes de $x^rM(x)$. R(x) por tener grado r-1 agrega entonces r coeficientes (bits) al mensaje representado por T(x) y es, en definitiva, el CRC del mensaje original M(x).

El receptor del mensaje modificado T(x) (o el sistema que lo recupera del almacenamiento) puede verificar la correctitud del mensaje comprobando que T(x) es dividible por G(x).

Lo más interesante de este método es que, a pesar de las apariencias, es sumamente sencillo de implementar, incluso en hardware, por tratarse de coeficientes binarios.

La elección del polinomio G(x) (denominado "generador") no es un tema menor, ya que de acuerdo a sus coeficientes tendrá mejores propiedades de detección. Los sistemas CRC que tienen polinomios G(x) con grado r y contienen término independiente pueden detectar errores de ráfagas de hasta r bits. Tener en cuenta que estos sistemas son muy buenos para ráfagas, pero no tanto para errores aislados si estos aparecen más de una vez en el mensaje.

Un ejemplo de polinomio que se usa es:

CRC-16-IBM -> $x^{16} + x^{15} + x^2 + 1$

que fue desarrollado originalmente por IBM para su protocolo de comunicaciones “Bisync”.

Otro ejemplo es:

CRC-16-CCITT -> $x^{16} + x^{12} + x^5 + 1$

desarrollado por la CCITT (actualmente UIT = Unión Internacional de Telecomunicaciones) para, entre otros, el protocolo de comunicaciones X.25.

También existen polinomios para calcular CRCs de 32 y 64 bits.