CAPÍTULO 2

RUMBO A LOS TURBO-CÓDIGOS: LA CODIFICACIÓN CONVOLUCIONAL

2.1. Introducción.

Hasta este momento, el término "codificación del canal" lo hemos empleado para referirnos a una estrategia de control de errores denominada *Forward Error Correction* (*FEC*), en la que se utilizan esquemas de codificación que son capaces tanto de detectar como de corregir errores en un proceso de transmisión de información. Sin embargo, ésta no es la única estrategia con la que un sistema de comunicación codificado puede contar para el control de errores. Existe otra estrategia conocida como *Automatic Repeat Request* (*ARQ*) en la que el nivel deseado de precisión en la información recibida se consigue mediante retransmisiones del mensaje enviado dado que se emplean esquemas de codificación que son capaces de detectar errores, pero no de corregirlos. De acuerdo con [LIN83], en un sistema ARQ cuando se detectan errores en el receptor se envía una solicitud de retransmisión del mensaje al transmisor, y esto se repite hasta que el mensaje es recibido correctamente por el receptor.

Debido a que el control de errores por medio de ARQ es una alternativa exclusiva para sistemas de comunicación que pueden soportar una transmisión de información en dos direcciones (de transmisor a receptor y viceversa), como un sistema half-dúplex o uno full-dúplex, generalmente son los sistemas símplex (en los que la transmisión de información se da únicamente de transmisor a receptor) los que ocupan FEC para el control de errores. Sin embargo, esto no quiere decir que sólo los sistemas símplex puedan emplear el esquema FEC, ya que este tipo de estrategia controladora de errores es también utilizada en la

práctica por varios sistemas de comunicación codificados full-dúplex como el sistema de telefonía celular. Dado que en esta Tesis trataremos exclusivamente con el esquema FEC, a continuación presentamos una breve descripción acerca del esquema ARQ.

Atendiendo a [LIN83], existen dos tipos de sistemas ARQ en general: ARQ-*stop-and-wait* y ARQ-*continuous*. En ARQ-*stop-and-wait*, el transmisor envía una palabra de código al receptor y espera una respuesta positiva (ACK) o negativa (NAK) de este último. Un ACK recibido por el transmisor significa que no se detectaron errores en la transmisión y por lo tanto se envía la siguiente palabra de código. Por el contrario, un NAK recibido por el transmisor implica que hubo errores en la transmisión y por lo tanto debe re-enviarse la palabra de código que fue transmitida. El esquema ARQ-*stop-and-wait* está diseñado para emplearse en canales half-dúplex.

En ARQ-*continuous* el transmisor envía palabras de código al receptor de manera constante y recibe respuestas de este último de la misma manera. Cuando un NAK es recibido, el transmisor debe iniciar una retransmisión que puede seguir uno de dos patrones: *go-back-N*-ARQ y *selective-repeat*-ARQ. El primer patrón de retransmisión consiste en re-enviar las palabras de código *desde* la que fue detectada errónea. *Selective-repeat*-ARQ consiste en retransmitir únicamente la palabra de código a la que corresponde el NAK recibido. Este último tipo de retransmisión es más eficiente que el primero, pero es más complejo en su implementación. El esquema ARQ-*continuous* está diseñado para utilizarse en canales full-dúplex.

La mayor ventaja de ARQ sobre FEC es que la detección de errores requiere un equipo de decodificación mucho menos complejo que el que requiere la corrección de errores. Otra desventaja que presenta FEC con respecto a ARQ es que con FEC puede darse el caso en que la secuencia decodificada binaria aún contenga algunos errores que serían ya

incorregibles. Sin embargo, en canales donde el ruido es persistente y por lo tanto se tiene una probabilidad de error alta (p.ej. en los sistemas de telefonía celular), emplear un esquema ARQ implicaría la necesidad de retransmitir varias veces una sola palabra de código, lo que limitaría de manera considerable la rapidez del sistema. En estos casos lo mejor es emplear un esquema FEC o una combinación de FEC para los patrones de error más frecuentes junto con ARQ para los patrones de error menos probables. Este tipo de combinación se conoce como *hybrid*-ARQ y a pesar de que no ha sido implementada en muchos sistemas, podría considerarse como la estrategia de control de errores más eficiente de acuerdo con [PET84].

Los esquemas de codificación más empleados por los sistemas ARQ son los *códigos de paridad* y los *códigos de redundancia cíclica* (*CRC – Cyclic Redundancy Check codes*), de los cuales puede encontrarse un análisis detallado en [McN82] [T.CAM04]. Es importante mencionar que a pesar de que estos códigos en realidad efectúan una *codificación para control de errores* (o *codificación del canal*), generalmente este término se utiliza en literatura científica en la misma forma en que se ha empleado y se empleará a lo largo de esta Tesis: para referirse a esquemas de codificación capaces de corregir errores.

2.2. Los esquemas de codificación FEC.

El campo de la codificación para control de errores cuenta con dos clases principales de códigos: *códigos de bloque* y *códigos convolucionales*.

La codificación convolucional juega un papel central en esta Tesis y por ende centraremos nuestra atención en este esquema de codificación, tratándolo de manera particular en el sub-capítulo 2.3. Sin embargo, es conveniente que antes presentemos una

breve descripción de la codificación de bloque para familiarizarnos con algunos conceptos que son utilizados ampliamente en la teoría de codificación del canal.

2.2.1. La codificación de bloque.

Supongamos un alfabeto de señalización para la transmisión binario (la salida del codificador del canal será binaria), que es el tipo de alfabeto que se considerará en esta Tesis mientras no se especifique lo contrario. El codificador para control de errores aceptará bits de información a la tasa de transferencia de información del sistema, R_S. Atendiendo a [LIN83] [MIC85], en la codificación de bloque el codificador divide la secuencia de bits de información en bloques de k bits cada uno llamados *mensajes*. Cada mensaje se representa por el *k-tuple* binario $\mathbf{u} = (u_1, u_2, \ldots, u_k)$, donde cada u_n es un 1 o un 0, y en total existen 2^k mensajes diferentes posibles. Para cada mensaje el codificador genera un *n-tuple* binario $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ llamado *palabra de código*, en el que $n > k$. De esta forma, el codificador es capaz de generar 2^k palabras de código distintas. El conjunto de estas 2^k palabras de código de longitud n se conoce como **código de bloque (n, k)**. El codificador producirá bits a una tasa igual a $R_C = R_S (n / k)$ bits por segundo.

Dado que cada palabra de código de n bits depende única y exclusivamente de un mensaje de entrada de k bits, se dice que el codificador de bloque *no tiene memoria* y puede ser implementado con un circuito lógico combinacional [LIN83]. La razón adimensional $R = k/n$ se conoce como *tasa de código* y la diferencia $n - k$ es el número de bits redundantes que el codificador añade a cada mensaje a transmitirse para el control de errores. De acuerdo con [T.ING98], tres importantes propiedades de un código de bloque las constituyen: el *peso de Hamming*, la *distancia de Hamming* y la *distancia mínima*.

Antes de definir cada uno de estos conceptos es necesario que definamos una importante sub-clase de los códigos de bloque en general, denominada *códigos de bloque lineales*. Atendiendo a [LIN83], un **código de bloque lineal (*n, k*)** es un código de bloque (*n*, *k*) en el que la suma en módulo-2 de cualesquiera dos palabras de código resulta también en una palabra de código. Recordemos que la suma en módulo-2 es equivalente a una operación lógica XOR. Los códigos Hamming fueron los primeros códigos de bloque lineales que se desarrollaron.

De acuerdo con [VAN89], la *distancia de Hamming* ***d*(x,y)** *entre dos palabras de código* **x** *y* **y** es el número de posiciones en las que difiere una de la otra.

Ejemplo 1. Distancia de Hamming entre dos palabras de código [VAN89].

Sean **x** y **y** dos palabras de código de longitud $n = 5$:

x = (**1**011**0**)

y = (**1**101**1**)

Las posiciones 1 y 4 (bits en negrita) son las únicas posiciones en que x e y son iguales, por lo que $d(x,y) = 3$.

La *distancia de Hamming* ***d*** *de un código de bloque C* es la *mínima distancia* de Hamming existente entre dos palabras de código distintas sobre todos los pares de palabras de código [VAN89]. Matemáticamente esto se expresa como:

$$d = \min \{d(x,y):\ x,y \in C, x \neq y\} \qquad (2.1)$$

Ejemplo 2. Distancia de Hamming de un código de bloque [VAN89].

Considérese el código de bloque $C = \{\mathbf{c}_0,\mathbf{c}_1,\mathbf{c}_2,\mathbf{c}_3\}$ donde:

$\mathbf{c}_0 = (00000)$

$\mathbf{c}_1 = (10110)$

$\mathbf{c}_2 = (01011)$

$\mathbf{c}_3 = (11101)$

Analizando todos los pares posibles de las palabras de código $\mathbf{c}_n$, se encuentra que el código C tiene una distancia de Hamming $d = 3$.

La distancia de Hamming de un código de bloque es un parámetro importante, ya que determina la capacidad de corrección ($\boldsymbol{t}$) y de detección ($\boldsymbol{e}$) de errores del código como [VAN89]:

$$t = (d - 1)/2 \tag{2.2}$$

$$e = d - 1 \tag{2.3}$$

Esto es, t y e definen el máximo número garantizado de errores corregidos o detectados, respectivamente, por palabra de código [STA04].

En comparación con la distancia de Hamming de un código de bloque, el *peso de Hamming de un código* está más bien definido con respecto a los códigos de bloque lineales. Sin embargo, estos dos conceptos en realidad pueden extenderse en su aplicación a otros tipos de códigos que no forzosamente tengan que ser códigos de bloque (para el caso de la distancia de Hamming) o códigos de bloque lineales (para el caso del peso de Hamming), como se verá en el Ejemplo 3.

Definimos el *peso de Hamming* ***w*(x)** *de un n-tuple binario* **x** como el número de elementos distintos de cero en **x**. Entonces, de acuerdo con [VAN89] el *peso de Hamming* ***w*(*C*)** *de un código de bloque lineal* ***C*** está dado por:

$$w(C) = \min \{w(x): \ x \in C, x \neq 0\} \tag{2.4}$$

Ejemplo 3. Peso de Hamming de un código de bloque lineal [VAN89].

Considérense los siguientes códigos de bloque lineales S_1 y S_2:

$S_1 = \{(0000), (1000), (0100), (1100)\}$

$S_2 = \{(0000), (1100), (0011), (1111)\}$

De acuerdo con la fórmula (2.4) podemos ver que en S_1 el peso de Hamming es $w(S_1) = 1$ y en S_2 el peso de Hamming es $w(S_2) = 2$. Si quisiéramos calcular la distancia de Hamming de S_1 y S_2 podríamos hacerlo empleando la ecuación (2.1), lo que nos daría por resultado que $d = 1$ para S_1, y $d = 2$ para S_2.

El que el peso de Hamming sea igual a la distancia de Hamming en cada código no es una coincidencia, ya que según [VAN89] en un código de bloque lineal siempre se cumple que $d = w(C)$.

2.3. La codificación convolucional.

De acuerdo con [LIN83], al igual que en la codificación de bloque, en la codificación convolucional el codificador acepta bloques de *k* bits de la secuencia de bits de información y produce una secuencia codificada de bloques de *n* bits (donde $n > k$) llamada *palabra de código*. Sin embargo, la principal diferencia con respecto a la codificación de bloque radica en que el codificador convolucional tiene memoria y por lo tanto sus *n* salidas en cualquier

unidad de tiempo dada no dependen únicamente de sus *k* entradas en esa unidad de tiempo, sino también de *m* bloques de entrada previos. Se dice entonces que el codificador convolucional tiene una *memoria de orden m*. El conjunto de secuencias codificadas producidas por un codificador de *k* entradas, *n* salidas, y memoria de orden *m* se conoce como ***código convolucional* (*n, k, m*)**.

La razón adimensional $R = k/n$ se conoce como *tasa de código* y la cantidad $n(m + 1)$ se conoce como *constraint length del código* [LIN83]. De esta última cantidad hablaremos posteriormente. Por lo general *k*, *n* y *m* son enteros pequeños: un código convolucional binario típico tiene $k = 1$, $n = 2$ ó 3, y $4 \leq m \leq 7$ [MIC85]. Al igual que el codificador de bloque, el codificador convolucional acepta bits de información a la tasa de transferencia de información del sistema, R_s, y produce bits a una tasa más alta.

Debido a que el codificador contiene memoria, debe ser implementado con un circuito lógico secuencial. El cómo utilizar la memoria para lograr una transmisión confiable de información por un canal ruidoso constituye el mayor problema en el diseño de un codificador convolucional.

De acuerdo con [LIN83], los códigos convolucionales fueron presentados por vez primera en [A.ELIA] en 1955 por P. Elias, como una alternativa a los códigos de bloque. Poco después, en [WOZ61] J. M. Wozencraft y B. Reiffen propusieron la decodificación secuencial como un esquema de decodificación eficiente para la codificación convolucional. En 1967 A. J. Viterbi propuso en [A.VITE] un esquema de decodificación de máxima probabilidad que era fácil de implementar para códigos con un orden bajo de memoria. Este esquema, llamado *decodificación de Viterbi*, junto con versiones mejoradas

de la decodificación secuencial llevaron a los códigos convolucionales a aplicaciones de comunicación satelital hacia 1970.

2.3.1. El codificador convolucional.

La Figura 2.1 nos muestra el codificador de un código convolucional (2, 1, 3). En la figura se puede observar que el codificador consta de un registro de desplazamiento de $m = 3$ etapas, junto con $n = 2$ sumadores módulo-2 y un multiplexor para poner en serie las salidas del codificador. Todos los codificadores convolucionales pueden ser implementados utilizando registros de desplazamiento multi-etapa de este tipo [LIN83].

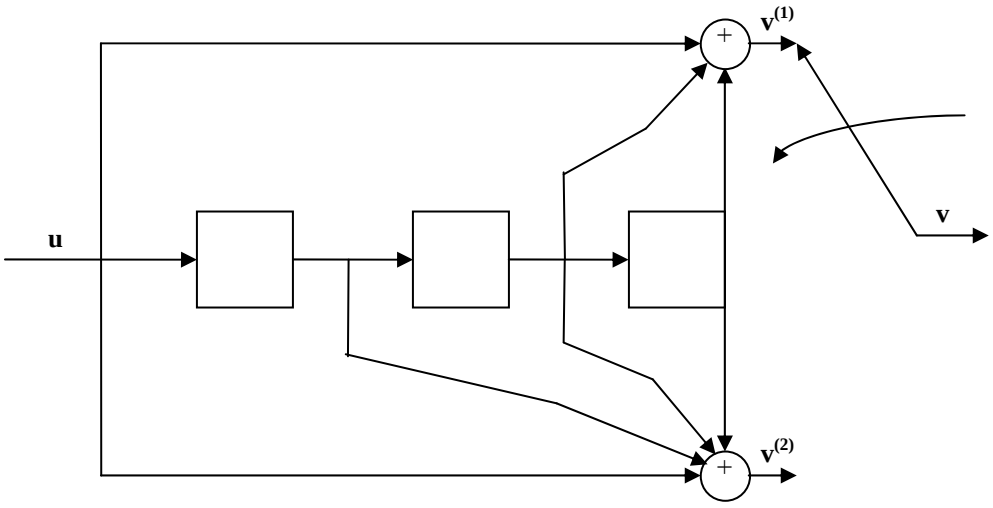

Figura 2.1. Codificador convolucional binario (2, 1, 3). [LIN83].

La secuencia de información $\mathbf{u} = (u_0, u_1, u_2, \ldots)$ entra al codificador un bit a la vez. El codificador es un sistema lineal y puede visualizarse como una forma de filtro digital con memoria, por lo que sus dos secuencias de salida $\mathbf{v}^{(1)} = (v_0^{(1)}, v_1^{(1)}, v_2^{(1)}, \ldots)$ y $\mathbf{v}^{(2)} = (v_0^{(2)}, v_1^{(2)}, v_2^{(2)}, \ldots)$ pueden obtenerse como la convolución discreta de la secuencia de entrada **u** con las dos *respuestas al impulso* del codificador. Las respuestas al impulso se

conocen como *secuencias generadoras del código* y se obtienen haciendo **u** = (1 0 0 . . .) y observando las dos secuencias de salida. El bit de la extrema izquierda de **u** se considera el primero en entrar al codificador y los registros de desplazamiento se suponen inicializados a cero. Dado que el codificador tiene una memoria de *m* unidades de tiempo, las respuestas al impulso pueden durar un máximo de *m*+1 unidades de tiempo y se representan como $\mathbf{g}^{(1)} = (g_0^{(1)}, g_1^{(1)}, \ldots, g_m^{(1)})$ y $\mathbf{g}^{(2)} = (g_0^{(2)}, g_1^{(2)}, \ldots, g_m^{(2)})$.

Para el codificador de la Figura 2.1, $\mathbf{g}^{(1)}$ = (1 0 1 1) y $\mathbf{g}^{(2)}$ = (1 1 1 1), y las ecuaciones de codificado son: $\mathbf{v}^{(1)} = \mathbf{u} * \mathbf{g}^{(1)}$ y $\mathbf{v}^{(2)} = \mathbf{u} * \mathbf{g}^{(2)}$. El símbolo $*$ indica convolución discreta, la cual implica que para toda $l \geq 0$:

$$v_l^{(j)} = \sum_{i=0}^{m} u_{l-i} g_i^{(j)} = u_l g_0^{(j)} + u_{l-1} g_1^{(j)} + \cdots + u_{l-m} g_m^{(j)}; \; j = 1,2,...,n \qquad (2.5)$$

donde todas las operaciones se efectúan en módulo-2 y $u_{l-i} = 0$ si $l < i$. Recordemos que una multiplicación en módulo-2 es equivalente a una multiplicación real entre 1´s y 0´s. Después del codificado, las dos secuencias de salida son multiplexadas para producir la secuencia llamada *palabra de código*. Esta secuencia está dada por:

$$\mathbf{v} = (v_0^{(1)} v_0^{(2)}, v_1^{(1)} v_1^{(2)}, v_2^{(1)} v_2^{(2)}, \ldots) \qquad (2.6)$$

Ejemplo 4. Generación de la palabra de código para un codificador (2,1,3) [LIN83].

Considérese la secuencia de información **u** = (1 0 1 1 1). Las secuencias de salida serán:

$\mathbf{v}^{(1)} = \mathbf{u} * \mathbf{g}^{(1)}$ = (1 0 1 1 1) * (1 0 1 1) = (1 0 0 0 0 0 0 1)

$\mathbf{v}^{(2)} = \mathbf{u} * \mathbf{g}^{(2)}$ = (1 0 1 1 1) * (1 1 1 1) = (1 1 0 1 1 1 0 1)

Empleando la ecuación (2.6) obtenemos la palabra de código buscada, que es:

v = (1 1, 0 1, 0 0, 0 1, 0 1, 0 1, 0 0, 1 1).

Es importante notar que la longitud de la palabra de código no es de 10 bits como intuitivamente podría pensarse al ser $k = 1$, $n = 2$ y la longitud de **u** = 5 bits. La longitud de la palabra de código es de 16 bits ya que el último bit de información distinto de 0 que entra al codificador tarda $m = 3$ unidades de tiempo más en desaparecer, lo que se traduce en $n(m) = 6$ bits adicionales a los diez bits generados por el codificador para los 5 bits de información. De esto se hablará un poco más formalmente más adelante.

2.3.1.1. La matriz generadora.

Si las secuencias generadoras $\mathbf{g}^{(1)}$ y $\mathbf{g}^{(2)}$ son entrelazadas y formamos el arreglo matricial siguiente:

$$\mathbf{G} = \begin{bmatrix} g_0^{(1)}g_0^{(2)} & g_1^{(1)}g_1^{(2)} & g_2^{(1)}g_2^{(2)} & \cdots & g_m^{(1)}g_m^{(2)} & & & \\ & g_0^{(1)}g_0^{(2)} & g_1^{(1)}g_1^{(2)} & \cdots & g_{m-1}^{(1)}g_{m-1}^{(2)} & g_m^{(1)}g_m^{(2)} & & \\ & & g_0^{(1)}g_0^{(2)} & \cdots & g_{m-2}^{(1)}g_{m-2}^{(2)} & g_{m-1}^{(1)}g_{m-1}^{(2)} & g_m^{(1)}g_m^{(2)} & \\ & & \vdots & & & & \vdots & \\ & & & & & & & \end{bmatrix}$$

donde todas las áreas blancas son ceros, entonces podemos expresar a la palabra de código **v** como:

$$\mathbf{v} = \mathbf{uG} \tag{2.7}$$

donde **u** es la secuencia de información y **G** se conoce como *matriz generadora del código*. La multiplicación matricial se realiza de la misma manera que una multiplicación de matrices con números reales, con la excepción de que las sumas y multiplicaciones se efectúan en módulo-2. Nótese que cada renglón de **G** es idéntico al anterior pero recorrido $n = 2$ lugares a la derecha. **G** tendrá el mismo número de renglones que la longitud de **u**.

Ejemplo 5. Generación de la palabra de código para un codificador (2,1,3) [LIN83].

Considérese nuevamente la secuencia de información **u** = (1 0 1 1 1). De acuerdo con la ecuación (2.7), la palabra de código **v** está dada por:

$$\mathbf{v} = (1\ 0\ 1\ 1\ 1) \begin{bmatrix} 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1 & & & & \\ & 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1 & & & \\ & & 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1 & & \\ & & & 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1 & \\ & & & & 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1 \end{bmatrix}$$

$$\mathbf{v} = (1\ 1, 0\ 1, 0\ 0, 0\ 1, 0\ 1, 0\ 1, 0\ 0, 1\ 1).$$

Sin hacer uso de la ecuación (2.7), el procedimiento para generar la palabra de código de un codificador convolucional se vuelve cada vez más complejo conforme k aumenta. Por ejemplo, para un codificador (3, 2, 1) la secuencia de información entra al codificador $k = 2$ bits a la vez, y puede representarse como $\mathbf{u} = (u_0^{(1)}u_0^{(2)}, u_1^{(1)}u_1^{(2)}, u_2^{(1)}u_2^{(2)}, \ldots)$ o bien como las dos secuencias de entrada $\mathbf{u}^{(1)} = (u_0^{(1)}, u_1^{(1)}, u_2^{(1)}, \ldots)$ y $\mathbf{u}^{(2)} = (u_0^{(2)}, u_1^{(2)}, u_2^{(2)}, \ldots)$. Existirán $n = 3$ secuencias generadoras para cada secuencia de entrada y por lo tanto las ecuaciones de codificado para las tres secuencias de salida serán más complejas que las obtenidas para el codificador (2,1,3) estudiado.

Sin embargo, la ecuación (2.7) puede aplicarse en general a cualquier codificador convolucional para obtener su palabra de código, y por lo tanto podemos plantear un procedimiento relativamente sencillo que nos permita obtener la palabra de código de cualquier tipo de codificador convolucional.

Atendiendo a [LIN83], en general podemos considerar que un codificador (*n, k, m*) contiene *k* registros de corrimiento multi-etapa, donde el número de etapas de cada uno es independiente y por tanto tal vez diferente al de los demás. Si *Ki* es el número de etapas (o longitud) del *i*-avo registro de corrimiento, entonces el orden de la memoria del codificador, *m*, queda definido como:

$$m = \max_{1 \le i \le k} Ki \tag{2.8}$$

que se interpreta como *la máxima longitud de entre todos los k registros de corrimiento*. La siguiente Figura nos muestra un ejemplo de un codificador convolucional (4,3,2) en el que las longitudes de los $k = 3$ registros de corrimiento son 0, 1 y 2.

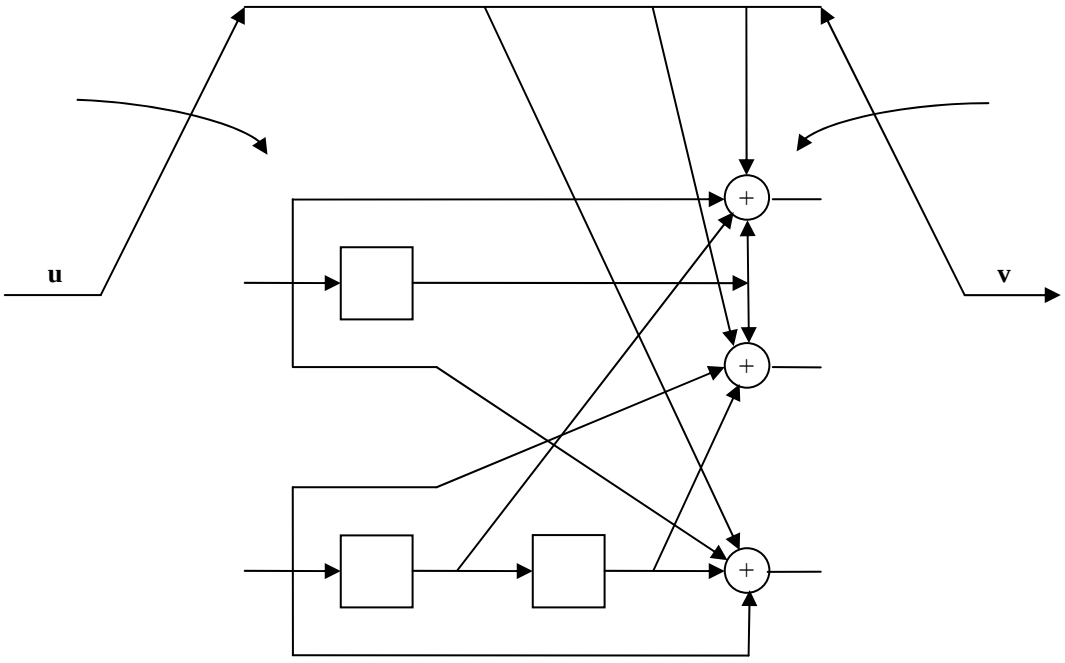

Figura 2.2. Codificador convolucional binario (4, 3, 2). [LIN83].

Para cualquier código convolucional (n, k, m), la matriz generadora del código está dada por [LIN83]:

$$\mathbf{G} = \begin{bmatrix} \mathbf{G_0} & \mathbf{G_1} & \mathbf{G_2} & \cdots & \mathbf{G_m} & & \\ & \mathbf{G_0} & \mathbf{G_1} & \cdots & \mathbf{G_{m-1}} & \mathbf{G_m} & \\ & & \mathbf{G_0} & \cdots & \mathbf{G_{m-2}} & \mathbf{G_{m-1}} & \mathbf{G_m} \\ & & \cdot & & & \cdot & \\ & & \cdot & & & \cdot & \\ & & \cdot & & & \cdot & \end{bmatrix}$$

donde cada $\mathbf{G}_l$ es una sub-matriz k x n cuyos componentes se definen a continuación.

Sea $\boldsymbol{g}_i^{(j)} = (g_{i,0}^{(j)}, g_{i,1}^{(j)}, \ldots, g_{i,m}^{(j)})$ la secuencia generadora correspondiente a una entrada $\boldsymbol{i}$ y a una salida $\boldsymbol{j}$ del codificador, donde cada secuencia generadora $\boldsymbol{g}_i^{(j)}$ se obtiene de hacer $\mathbf{u}$ = (1 0 0 0 . . .) en la entrada $\boldsymbol{i}$ del codificador (mientras las otras entradas se suponen cero) y observar la salida $\boldsymbol{j}$ del mismo. Por cada entrada que tenga el codificador existirán tantas secuencias generadoras como salidas tenga el codificador, resultando en un total de k x n secuencias generadoras del código. Estas secuencias tienen una duración de m+1 unidades de tiempo cada una, como se puede observar de la definición de $\boldsymbol{g}_i^{(j)}$. Entonces, cada sub-matriz $\mathbf{G}_l$ se forma de la siguiente manera [LIN83]:

$$\mathbf{G}_l = \begin{bmatrix} g_{1,l}^{(1)} & g_{1,l}^{(2)} & \cdots & g_{1,l}^{(n)} \\ g_{2,l}^{(1)} & g_{2,l}^{(2)} & \cdots & g_{2,l}^{(n)} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ g_{k,l}^{(1)} & g_{k,l}^{(2)} & \cdots & g_{k,l}^{(n)} \end{bmatrix}$$

Es importante notar que cada conjunto de k renglones de **G** es idéntico al anterior pero desplazado n lugares a la derecha.

De este modo, para una secuencia de información $\mathbf{u} = (\mathbf{u_0}, \mathbf{u_1}, \ldots) = (u_0^{(1)}u_0^{(2)}\ldots u_0^{(k)}, u_1^{(1)}u_1^{(2)}\ldots u_1^{(k)}, \ldots)$, la palabra de código $\mathbf{v} = (\mathbf{v_0}, \mathbf{v_1}, \ldots) = (v_0^{(1)}v_0^{(2)}\ldots v_0^{(n)}, v_1^{(1)}v_1^{(2)}\ldots v_1^{(n)}, \ldots)$ está dada por $\mathbf{v} = \mathbf{uG}$, donde **G** deberá tener tantos renglones como bits tenga **u** [LIN83].

Anteriormente definimos a la cantidad $n_A = n(m + 1)$ como *constraint length* de un código convolucional (n, k, m). Dado que cada bit de información puede permanecer en un codificador convolucional hasta $m + 1$ unidades de tiempo, y que en cada unidad de tiempo ese bit puede afectar a cualquiera de las n salidas del codificador, n_A representa el máximo número de salidas del codificador que pueden ser afectadas por un solo bit de información [LIN83]. Las constraint lengths de los códigos de las Figuras 2.1 y 2.2 son 8 y 12 respectivamente.

Como habremos podido observar en los Ejemplos 4 y 5, para una secuencia de s bits de entrada, un codificador convolucional produce una secuencia de salida mayor a $n(s/k)$ bits. Esto se debe a que en general, atendiendo a [LIN83], para una secuencia de información de longitud finita $k \bullet L$, la palabra de código correspondiente tiene una longitud de $n(L + m)$ bits ya que, para limpiar la memoria del codificador, se generan las últimas $n \bullet m$ salidas por medio de bloques de ceros luego de que el último bloque de información entra al codificador.

2.3.1.2. Los polinomios generadores y la matriz de función de transferencia.

Dado que la palabra de código **v** es una combinación lineal de renglones de la matriz generadora **G**, un código convolucional (n, k, m) es un código lineal. En cualquier sistema

lineal, todas las operaciones en el dominio del tiempo que involucren a la convolución pueden reemplazarse por operaciones en el dominio de una transformada que impliquen una multiplicación polinomial [LIN83]. En un codificador convolucional, cada secuencia en las ecuaciones de codificado puede ser reemplazada por un polinomio correspondiente, y la operación de convolución puede ser reemplazada por una multiplicación polinómica. Para expresar esto en forma más clara, consideremos el siguiente Ejemplo.

Ejemplo 6. Transformación de las ecuaciones de codificado de un codificador (2,1,*m*) [LIN83].

Consideremos un codificador convolucional (2,1,*m*). Las ecuaciones de codificado para este codificador están dadas por: $\mathbf{v}^{(1)} = \mathbf{u} * \mathbf{g}^{(1)}$ y $\mathbf{v}^{(2)} = \mathbf{u} * \mathbf{g}^{(2)}$. Estas ecuaciones están en el dominio del tiempo e implican una convolución, por lo que conviene transformarlas de la siguiente manera:

$$\mathbf{v}^{(1)} = \mathbf{u} * \mathbf{g}^{(1)} \rightarrow \mathbf{v}^{(1)}(\mathbf{D}) = \mathbf{u}(\mathbf{D})\,\mathbf{g}^{(1)}(\mathbf{D})$$

$$\mathbf{v}^{(2)} = \mathbf{u} * \mathbf{g}^{(2)} \rightarrow \mathbf{v}^{(2)}(\mathbf{D}) = \mathbf{u}(\mathbf{D})\,\mathbf{g}^{(2)}(\mathbf{D})$$

donde: $\mathbf{u}(\mathbf{D}) = u_0 + u_1\mathbf{D} + u_2\mathbf{D}^2 + \ldots$ es la secuencia de información.

$\mathbf{v}^{(1)}(\mathbf{D}) = v_0^{(1)} + v_1^{(1)}\mathbf{D} + v_2^{(1)}\mathbf{D}^2 + \ldots$
$\mathbf{v}^{(2)}(\mathbf{D}) = v_0^{(2)} + v_1^{(2)}\mathbf{D} + v_2^{(2)}\mathbf{D}^2 + \ldots$ son las secuencias codificadas.

$\mathbf{g}^{(1)}(\mathbf{D}) = g_0^{(1)} + g_1^{(1)}\mathbf{D} + \ldots + g_m^{(1)}\mathbf{D}^m$
$\mathbf{g}^{(2)}(\mathbf{D}) = g_0^{(2)} + g_1^{(2)}\mathbf{D} + \ldots + g_m^{(2)}\mathbf{D}^m$ son los **polinomios generadores**.

La palabra de código final está dada por: $\mathbf{v}(\mathbf{D}) = \mathbf{v}^{(1)}(\mathbf{D}^2) + \mathbf{D}\mathbf{v}^{(2)}(\mathbf{D}^2)$.

Las multiplicaciones y sumas polinomiales se efectúan al igual que en el álgebra convencional, pero consierando que las sumas y multiplicaciones entre coeficientes se realizan en módulo-2.

El operador indefinido "D" puede interpretarse como un *operador de retardo*, donde su exponente indica el número de unidades de tiempo que un bit está retrasado con respecto al bit inicial en la secuencia en cuestión.

Debido a que el codificador convolucional es un sistema lineal, si $\mathbf{u}^{(i)}\mathbf{(D)}$ representa a la *i*-ava secuencia de entrada y $\mathbf{v}^{(j)}\mathbf{(D)}$ representa a la *j*-ava secuencia de salida, el polinomio generador $\mathbf{g}_i^{(j)}\mathbf{(D)}$ puede interpretarse como la *función de transferencia del codificador entre la entrada i y la salida j* [LIN83]. Como en cualquier sistema lineal de *k* entradas y *n* salidas, existirán un total de *k* x *n* funciones de transferencia en un codificador convolucional. Estas funciones de transferencia pueden representarse por la *matriz de función de transferencia* [LIN83]:

$$\mathbf{G(D)} = \begin{bmatrix} \mathbf{g}_1^{(1)}\mathbf{(D)} & \mathbf{g}_1^{(2)}\mathbf{(D)} & \dots & \mathbf{g}_1^{(n)}\mathbf{(D)} \\ \mathbf{g}_2^{(1)}\mathbf{(D)} & \mathbf{g}_2^{(2)}\mathbf{(D)} & \dots & \mathbf{g}_2^{(n)}\mathbf{(D)} \\ \vdots & \vdots & & \vdots \\ \mathbf{g}_k^{(1)}\mathbf{(D)} & \mathbf{g}_k^{(2)}\mathbf{(D)} & \dots & \mathbf{g}_k^{(n)}\mathbf{(D)} \end{bmatrix}$$

Las ecuaciones de codificado para un código convolucional (*n*, *k*, *m*) pueden expresarse, entonces, como [LIN83]:

$$\mathbf{V(D)} = \mathbf{U(D)G(D)} \tag{2.9}$$

donde:

- $\mathbf{U(D)} = [\mathbf{u}^{(1)}\mathbf{(D)}, \mathbf{u}^{(2)}\mathbf{(D)}, \dots, \mathbf{u}^{(k)}\mathbf{(D)}]$ es el *k*-tuple de secuencias de entrada, y
- $\mathbf{V(D)} = [\mathbf{v}^{(1)}\mathbf{(D)}, \mathbf{v}^{(2)}\mathbf{(D)}, \dots, \mathbf{v}^{(n)}\mathbf{(D)}]$ es el *n*-tuple de secuencias de salida.

La palabra de código final estará dada por:

$$\mathbf{v(D)} = \mathbf{v}^{(1)}\mathbf{(D}^n\mathbf{)} + \mathbf{D}\mathbf{v}^{(2)}\mathbf{(D}^n\mathbf{)} + \dots + \mathbf{D}^{n-1}\mathbf{v}^{(n)}\mathbf{(D}^n\mathbf{)} \tag{2.10}$$

Prácticamente con la ecuación (2.10) podemos encontrar en una manera relativamente sencilla la palabra de código producida por cualquier codificador convolucional.

Ejemplo 7. Empleo de la ecuación (2.10) en el codificador de la Figura 2.1.

Consideremos el codificador convolucional (2, 1, 3) mostrado en la Figura 2.1. Las secuencias generadoras para este código son $\mathbf{g}^{(1)} = (1\ 0\ 1\ 1)$ y $\mathbf{g}^{(2)} = (1\ 1\ 1\ 1)$, por lo que sus polinomios generadores son:

$\mathbf{g}^{(1)}(\mathbf{D}) = 1 + \mathbf{D}^2 + \mathbf{D}^3$

$\mathbf{g}^{(2)}(\mathbf{D}) = 1 + \mathbf{D} + \mathbf{D}^2 + \mathbf{D}^3$

Si consideramos que la secuencia de información está dada por $\mathbf{u} = (1\ 0\ 1\ 1\ 1)$, entonces:

$\mathbf{U(D)} = [\mathbf{u}^{(1)}(\mathbf{D})] = [(1 + \mathbf{D}^2 + \mathbf{D}^3 + \mathbf{D}^4)]$

La matriz de función de transferencia para este código es la siguiente:

$\mathbf{G(D)} = [\mathbf{g}_1{}^{(1)}(\mathbf{D})\ \ \mathbf{g}_1{}^{(2)}(\mathbf{D})] = [\mathbf{g}^{(1)}(\mathbf{D})\ \mathbf{g}^{(2)}(\mathbf{D})] = [(1 + \mathbf{D}^2 + \mathbf{D}^3)\ (1 + \mathbf{D} + \mathbf{D}^2 + \mathbf{D}^3)]$

De acuerdo con la ecuación (2.9) $\mathbf{V(D)} = \mathbf{U(D)G(D)}$, por lo que:

$\mathbf{V(D)} = [(1 + \mathbf{D}^2 + \mathbf{D}^3 + \mathbf{D}^4)]\ [(1 + \mathbf{D}^2 + \mathbf{D}^3)\ (1 + \mathbf{D} + \mathbf{D}^2 + \mathbf{D}^3)]$

$= [\{(1 + \mathbf{D}^2 + \mathbf{D}^3 + \mathbf{D}^4)(\ 1 + \mathbf{D}^2 + \mathbf{D}^3)\}\{(1 + \mathbf{D}^2 + \mathbf{D}^3 + \mathbf{D}^4)(\ 1 + \mathbf{D} + \mathbf{D}^2 + \mathbf{D}^3)\}]$

Como podemos ver, la multiplicación matricial se realiza siguiendo las reglas convencionales para multiplicación de matrices. Efectuando la multiplicación de polinomios dentro de cada par de llaves y reduciendo términos semejantes, todo en módulo-2, llegamos a:

$\mathbf{V(D)} = [\{1 + \mathbf{D}^7\}\ \{1 + \mathbf{D} + \mathbf{D}^3 + \mathbf{D}^4 + \mathbf{D}^5 + \mathbf{D}^7\}]$

$\mathbf{v}^{(1)}(\mathbf{D}) = 1 + \mathbf{D}^7$

$\mathbf{v}^{(2)}(\mathbf{D}) = 1 + \mathbf{D} + \mathbf{D}^3 + \mathbf{D}^4 + \mathbf{D}^5 + \mathbf{D}^7$

De acuerdo con la ecuación (2.10), la palabra de código producida por el codificador de la Figura 2.1 para la secuencia de entrada **u** es:

$$\mathbf{v}(\mathbf{D}) = \mathbf{v}^{(1)}(\mathbf{D}^n) + \mathbf{D}\mathbf{v}^{(2)}(\mathbf{D}^n) + \ldots + \mathbf{D}^{n-1}\mathbf{v}^{(n)}(\mathbf{D}^n)$$

$$= 1 + (\mathbf{D}^2)^7 + \mathbf{D}(1 + \mathbf{D}^2 + (\mathbf{D}^2)^3 + (\mathbf{D}^2)^4 + (\mathbf{D}^2)^5 + (\mathbf{D}^2)^7)$$

$$= 1 + \mathbf{D}^{14} + \mathbf{D} + \mathbf{D}^3 + \mathbf{D}^7 + \mathbf{D}^9 + \mathbf{D}^{11} + \mathbf{D}^{15}$$

$$= 1 + \mathbf{D} + \mathbf{D}^3 + \mathbf{D}^7 + \mathbf{D}^9 + \mathbf{D}^{11} + \mathbf{D}^{14} + \mathbf{D}^{15}$$

Esta expresión en el dominio del tiempo es igual a (1 1 0 1 0 0 0 1 0 1 0 1 0 0 1 1), que es idéntico al 16-tuple binario **v** obtenido en los Ejemplos 4 y 5.

2.3.1.3. El diagrama de estados del codificador.

Debido a que un codificador convolucional es un circuito secuencial, su operación puede ser descrita por un *diagrama de estados*. El *estado del codificador* se define como el contenido de sus registros de corrimiento. Como ya se mencionó anteriormente, en general se puede considerar que un codificador (*n, k, m*) contiene *k* registros de corrimiento multi-etapa, donde el número de etapas entre cada uno de ellos puede variar. De este modo, para un código con $k > 1$ el *i*-avo registro de corrimiento contiene *Ki* bits de información previos. Si definimos la *memoria total del codificador K* como la suma de las longitudes *Ki* de los *k* registros de corrimiento del codificador, el *estado del codificador en la unidad de tiempo l* será el *K*-tuple binario: $(u_{l-1}^{(1)} u_{l-2}^{(1)} \ldots u_{l-K1}^{(1)} u_{l-1}^{(2)} u_{l-2}^{(2)} \ldots u_{l-K2}^{(2)} \ldots u_{l-1}^{(k)} \ldots u_{l-Kk}^{(k)})$, donde $u_l^{(1)}\ u_l^{(2)}\ u_l^{(3)} \ldots\ u_l^{(k)}$ son las entradas del codificador en esa unidad de tiempo [LIN83].

Si un codificador convolucional tiene $k = 1$, entonces $K = K_1 = m$ y el estado del codificador en la unidad de tiempo l será (u_{l-1} u_{l-2} ... u_{l-m}). Para poder interpretar el diagrama de estados de un codificador convolucional en general consideremos como ejemplo al siguiente diagrama, el cual corresponde a nuestro codificador convolucional (2,1,3) de la Figura 2.1.

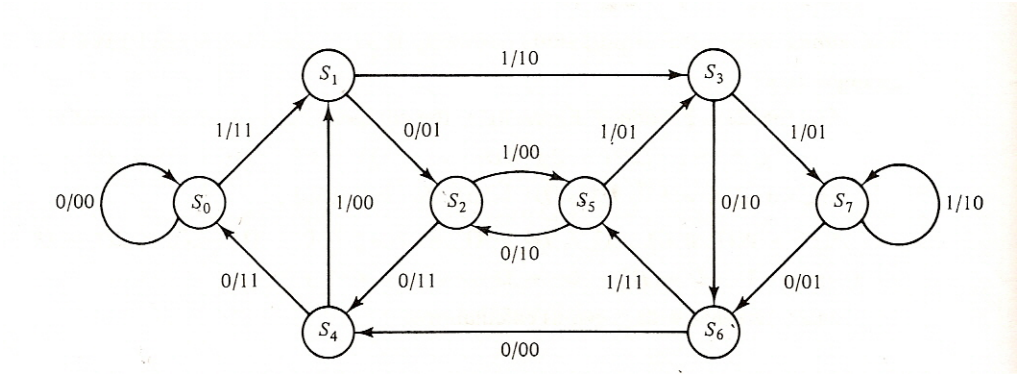

Figura 2.3. Diagrama de estados del codificador (2, 1, 3). [LIN83].

Dentro de los círculos se representan los estados del codificador en la unidad de tiempo l denotados por S_x, donde la "x" es el correspondiente número decimal de la representación binaria (K-tuple) invertida del estado en cuestión. Así, para el código (2,1,3) el estado S_6 indica que en el registro de corrimiento del codificador se tendrá el tuple (0 1 1), que visto de derecha a izquierda corresponde al 6 decimal. Los números en las flechas indican: *los k bits de entrada / los n bits producidos*. De esta forma, partiendo del estado S_0 (registros de corrimiento inicializados a cero) y siguiendo la trayectoria determinada por la secuencia de información a través del diagrama de estados, podemos obtener la palabra de código correspondiente a cualquier secuencia de información dada.

Para nuestro ejemplo, si consideramos que la secuencia de entrada **u** = (1 1 1 0 1), a partir de la Figura 2.3 podemos observar que para obtener la palabra de código la

trayectoria a seguir es: S_0 S_1 S_3 S_7 S_6 S_5. Al seguir esta trayectoria se observa que la secuencia de salida es: 1 1 1 0 0 1 0 1 1 1. Para limpiar el registro de corrimiento de nuestro codificador es necesario introducir $m = 3$ bloques de $k = 1$ bits de valor 0 después del último bloque de información distinto de cero que entró al codificador. En este caso partimos de S_5 y observamos que al recorrer la trayectoria S_5 S_2 S_4 S_0 obtenemos el resto de la palabra de código buscada, que finalmente resulta ser: 1 1 1 0 0 1 0 1 1 1 1 0 1 1 1 1.

2.3.1.4. Métricas de distancia para códigos convolucionales.

El desempeño de un código convolucional depende del algoritmo de decodificación que se emplee y de las propiedades de distancia del código. La medida de distancia más importante para códigos convolucionales la constituye la *mínima distancia libre* definida como [LIN83]:

$$d_{free} = \min \{d(\mathbf{v}', \mathbf{v}''): \mathbf{u}' \neq \mathbf{u}''\} \tag{2.11}$$

donde $\mathbf{v}'$ y $\mathbf{v}''$ son las palabras de código correspondientes a las secuencias de información $\mathbf{u}'$ y $\mathbf{u}''$ respectivamente. La ecuación (2.11) nos dice que la mínima distancia libre es igual a la mínima distancia de Hamming (ver Ejemplo 1) existente entre cualesquiera dos palabras de código en un código convolucional. En esta ecuación se asume que si $\mathbf{u}'$ y $\mathbf{u}''$ son de longitudes distintas, la secuencia más corta es aumentada con ceros de forma tal que sus palabras de código correspondientes sean de tamaños iguales.

Otra forma de representar a la mínima distancia libre es en términos del peso de Hamming de un n-tuple binario, definido en la sección 2.1.1. De este modo [LIN83]:

$$\begin{aligned} d_{free} &= \min \{w(\mathbf{v}' + \mathbf{v}''): \mathbf{u}' \neq \mathbf{u}''\} \\ &= \min \{w(\mathbf{v}): \mathbf{u} \neq \mathbf{0}\} \\ &= \min \{w(\mathbf{uG}): \mathbf{u} \neq \mathbf{0}\} \end{aligned} \tag{2.12}$$

donde **v** es la palabra de código correspondiente a la secuencia de información **u**, y **G** es la matriz generadora del código. Así, d_{free} es el mínimo peso existente entre las palabras de código de cualquier longitud producidas por una secuencia de información distinta de cero.

Otra medida de distancia importante para códigos convolucionales es la *función de distancia de columna* (*CDF*, por sus siglas en inglés de *Column Distance Function*). Si hacemos que $[\mathbf{v}]i = (v_0^{(1)}v_0^{(2)}\ldots\ v_0^{(n)},\ v_1^{(1)}v_1^{(2)}\ldots\ v_1^{(n)},\ \ldots\ ,\ v_i^{(1)}v_i^{(2)}\ldots\ v_i^{(n)})$ denote la *truncación i-ava* de la palabra de código **v**, y $[\mathbf{u}]i = (u_0^{(1)}u_0^{(2)}\ldots\ u_0^{(k)},\ u_1^{(1)}u_1^{(2)}\ldots\ u_1^{(k)},\ \ldots\ ,\ u_i^{(1)}u_i^{(2)}\ldots\ u_i^{(k)})$ denote la *truncación i-ava* de la secuencia de información **u**, la *CDF de orden i* se define como:

$$\begin{aligned} d_i &= \min\ \{d([\mathbf{v}']i, [\mathbf{v}'']i)\colon [\mathbf{u}']0 \neq [\mathbf{u}'']0\} \\ &= \min\ \{w[\mathbf{v}]i\colon [\mathbf{u}]0 \neq \mathbf{0}\} \end{aligned} \qquad (2.13)$$

donde **v**, **v´**, y **v´´** nuevamente son las palabras de código correspondientes a las secuencias de información **u**, **u´** y **u´´**, respectivamente. De esta forma, d_i es el mínimo peso existente sobre las primeras $(i + 1)$ unidades de tiempo de entre las palabras de código producidas por una secuencia de información cuyo bloque inicial no es cero. Si $i = m$ (el orden de la memoria de un codificador convolucional), $d_i = d_m = d_{min}$. Esta última constituye otra medida de distancia de interés para codificación convolucional, y se conoce como *distancia mínima* de un código convolucional.

2.3.2. La decodificación convolucional.

La función idónea de todo decodificador para control de errores es encontrar de entre el conjunto de todas las palabras posibles que pudieran transmitirse la palabra de código que más se asemeje a la información recibida [MIC85], es decir, la palabra de código que minimice la probabilidad de error del decodificador. Esto se conoce como *decodificación*

de máxima probabilidad y resulta ser una estrategia de decodificación óptima. En palabras de Peter Sweeney en [SWE91]: "En principio, la mejor forma de decodificar en contra de errores aleatorios es comparar la secuencia recibida con cada secuencia de código posible". (Sweeney, 1991: 118). De acuerdo con [ADÁ91], "la idea básica del decodificado es la misma para códigos lineales y convolucionales: recibida una palabra, la decodificamos como la palabra de código de máxima probabilidad, esto es, como la palabra de código que tenga la mínima distancia de Hamming con respecto a la palabra recibida". (Adámek, 1991: 279).

Hace más de cuatro décadas, sin embargo, el decodificado de máxima probabilidad no parecía ser un esquema de decodificación deseable para códigos convolucionales debido a la gran carga computacional que implicaba el tener que hacer búsquedas en el espacio completo de un código. No obstante, en 1967 Viterbi introdujo un algoritmo de decodificación de máxima probabilidad para códigos convolucionales que resultó ser práctico para códigos con una constraint-length baja: el *algoritmo de Viterbi* [LIN83] [MIC85]. Este algoritmo hace uso de la estructura altamente repetitiva del árbol del código (otra forma de representación gráfica del proceso de codificado para códigos convolucionales) para reducir en buena medida la cantidad de operaciones requeridas para hacer búsquedas en todo el espacio del código [MIC85].

A pesar de esto, como ya se mencionó el algoritmo de Viterbi tenía la restricción de ser una buena opción para el decodificado de códigos convolucionales con una constraint-length baja, lo que llevó a los investigadores en esta rama a tratar de encontrar otros métodos de decodificación convolucional que superaran la eficiencia del algoritmo de Viterbi al poderse aplicar de manera práctica a códigos convolucionales con constraint-length´s bajas y altas. Hoy en día se cuenta con tres métodos principales para

decodificación convolucional, cada uno de ellos con sus propias ventajas y desventajas que los hacen deseables para distintos tipos de aplicaciones. Estos métodos son: el algoritmo de Viterbi, la Decodificación Secuencial, y la Decodificación de Umbral.

Para los propósitos de esta Tesis trataremos únicamente el algoritmo de decodificación de Viterbi. La temática relacionada con decodificación secuencial y decodificación de umbral puede ser consultada en [LIN83] [MIC85] [PET84].

2.3.2.1. El algoritmo de Viterbi.

Si expandemos el diagrama de estados de un codificador convolucional en el tiempo, la estructura que resulta se conoce como *diagrama de Trellis*. La siguiente Figura nos muestra el diagrama de Trellis para un codificador convolucional (3,1,2) con matriz de función de transferencia $\mathbf{G(D)} = [1 + \mathbf{D}, 1 + \mathbf{D}^2, 1 + \mathbf{D} + \mathbf{D}^2]$ y una secuencia de información de tamaño $kL = 5$ bits.

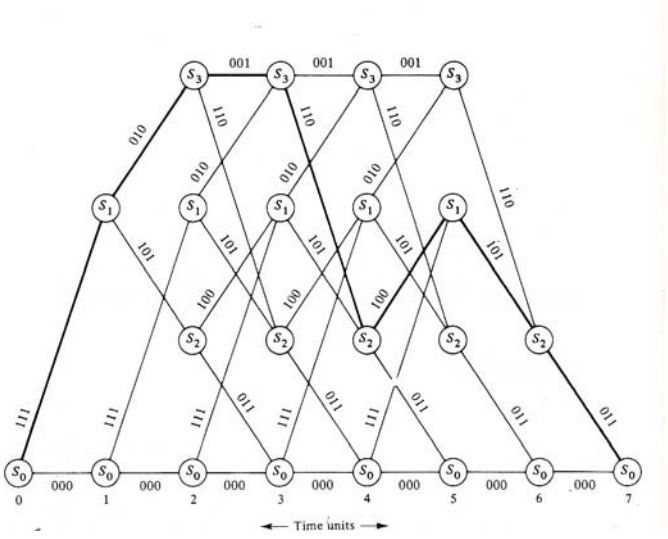

Figura 2.4. Diagrama de Trellis para el código (3,1,2) con $L = 5$. [LIN83].

Un diagrama de Trellis contiene $L + m + 1$ unidades de tiempo (o niveles) para un codificador (n, k, m). Para el codificador (3,1,2) considerado, $L = 5$ y $m = 2$ por lo que su

diagrama de Trellis consta de 8 niveles como se puede apreciar en la Figura 2.4. Como podremos observar en este diagrama, de cada uno de los distintos estados del codificador (los cuales están representados en los círculos) salen dos ramas. La rama superior que abandona a cada estado en la unidad de tiempo *i* corresponde a una entrada al codificador $u = 1$ en esa unidad de tiempo; de manera contraria, la rama inferior corresponde a una entrada $u = 0$ en la unidad de tiempo considerada. Cada una de estas ramas está etiquetada con las *n* salidas correspondientes producidas por el codificador.

De acuerdo con [LIN83], en general para un codificador (*n, k, m*) con una secuencia de información de tamaño *kL* existirán, en su diagrama de Trellis, 2^k ramas saliendo de cada estado y 2^{kL} trayectorias distintas a través del Trellis correspondientes a las 2^{kL} palabras de código posibles. Cada una de estas palabras de código tendrá una longitud de $N = n(L + m)$ bits. En la Figura 2.4 se puede observar resaltada en negrita la trayectoria de la palabra de código correspondiente a la secuencia de información **u** = (1 1 1 0 1) para el codificador considerado. Siguiendo dicha trayectoria del nivel 0 al 7 podemos ver que esa palabra de código es: **v** = (1 1 1, 0 1 0, 0 0 1, 1 1 0, 1 0 0, 1 0 1, 0 1 1).

Para poder comprender el algoritmo de Viterbi, es necesario que primero detallemos formalmente el significado del concepto "decodificación de máxima probabilidad". Para simplificar las cosas definimos un *canal discreto sin memoria* como un bloque virtual que engloba al Modulador, al Canal AWGN y al Demodulador de la Figura 1.5. Consideremos que una secuencia de información $\mathbf{u} = (\mathbf{u_0}, \ldots, \mathbf{u_{L-1}}) = (u_0, u_1, \ldots, u_{kL-1})$ de longitud *kL* es codificada en una palabra de código binaria $\mathbf{v} = (\mathbf{v_0}, \ldots, \mathbf{v_{L+m-1}}) = (v_0, v_1, \ldots, v_{N-1})$ de longitud $N = n(L + m)$ y que una secuencia Q-aria $\mathbf{r} = (\mathbf{r_0}, \ldots, \mathbf{r_{L+m-1}}) = (r_0, r_1, \ldots, r_{N-1})$ es recibida a la salida de un canal discreto sin memoria. Como ya se discutió anteriormente,

un decodificador para control de errores debe producir un estimado **v´** de la palabra de código **v** basado en la secuencia recibida **r**. Un *decodificador de máxima probabilidad* es aquel que elige **v´** como la palabra de código **v** que maximiza la función log-likelihood dada por [LIN83]:

$$\log P(\mathbf{r} \mid \mathbf{v}) = \sum_{i=0}^{L+m-1} \log P(\mathbf{r_i} \mid \mathbf{v_i}) = \sum_{i=0}^{N-1} \log P(r_i \mid v_i) \tag{2.14}$$

donde $P(r_i \mid v_i)$ es la probabilidad de que el símbolo r_i sea recibido a la salida del canal discreto sin memoria dado que el símbolo v_i fue enviado a través de dicho canal. De acuerdo con [LIN83] [MIC85], este tipo de probabilidades, conocidas como *probabilidades de transición de un canal discreto sin memoria*, pueden calcularse a partir de un conocimiento de las señales utilizadas, de la distribución de probabilidad del ruido, y del umbral de cuantización de la salida del demodulador.

Atendiendo a [LIN83], la función log-likelihood ($\log P(\mathbf{r} \mid \mathbf{v})$) se conoce como la *métrica asociada con la trayectoria* **v**, y se denota como $M(\mathbf{r} \mid \mathbf{v})$. Los términos $\log P(\mathbf{r_i} \mid \mathbf{v_i})$ en la sumatoria de la ecuación (2.14) se conocen como *métricas de rama* y se denotan como $M(\mathbf{r_i} \mid \mathbf{v_i})$, mientras que los términos $\log P(r_i \mid v_i)$ se conocen como *métricas de bit* y se denotan como $M(r_i \mid v_i)$. De este modo, la ecuación (2.14) puede reescribirse como:

$$M(\mathbf{r} \mid \mathbf{v}) = \sum_{i=0}^{L+m-1} M(\mathbf{r_i} \mid \mathbf{v_i}) = \sum_{i=0}^{N-1} M(r_i \mid v_i) \tag{2.15}$$

Una *métrica de trayectoria parcial* para las primeras *j* ramas de una trayectoria puede escribirse como:

$$M([\mathbf{r} \mid \mathbf{v}]_j) = \sum_{i=0}^{j-1} M(\mathbf{r_i} \mid \mathbf{v_i}) = \sum_{i=0}^{nj-1} M(r_i \mid v_i) \tag{2.16}$$

Ahora sí estamos listos para enunciar el algoritmo de Viterbi, el cual recibida una secuencia **r** de un canal discreto sin memoria encuentra la trayectoria de mayor $M(\mathbf{r} \mid \mathbf{v})$ a través del diagrama de Trellis del código que se considere. De acuerdo con [LIN83], el algoritmo de Viterbi es el siguiente:

- *Paso 1.* En el diagrama de Trellis, comenzando en la unidad de tiempo $j = m$ calcular la métrica de trayectoria parcial de la única trayectoria que entra a cada estado y que comienza en el nivel 0. Almacenar la trayectoria (llamada *sobreviviente*) junto con su métrica para cada estado.
- *Paso 2.* Incrementar j en 1. Calcular la métrica parcial para todas las trayectorias que entran a un estado añadiendo la métrica de la rama que entra a ese estado a la métrica de la trayectoria sobreviviente en la unidad de tiempo anterior que se conecta a esa rama. Para cada uno de los estados, almacenar la trayectoria con la métrica más alta (trayectoria sobreviviente) junto con su métrica, y eliminar todas las otras trayectorias.
- *Paso 3.* Si $j < L + m$, repetir el Paso 2. De otra manera, parar.

La trayectoria buscada será la trayectoria sobreviviente del nivel 0 al nivel $L + m$ a través del diagrama de Trellis considerado.

Generalmente es más conveniente y sencillo utilizar enteros positivos como métricas en vez de las métricas de bit reales. De acuerdo con [LIN83], una métrica de bit $M(r_i \mid v_i)$ puede reemplazarse por la ecuación (2.17) sin afectar el desempeño del algoritmo de Viterbi. La constante c_1 puede ser cualquier número real y la constante c_2 puede ser cualquier número real positivo.

$$M(r_i \mid v_i) = c_2[\log P(r_i \mid v_i) + c_1] \tag{2.17}$$

Para ejemplificar el algoritmo de Viterbi, consideremos el siguiente Ejemplo.

Ejemplo 8. Empleo del algoritmo de Viterbi para decodificación convolucional [LIN83].

Supongamos un canal discreto sin memoria **A** de entrada binaria y salida cuaternaria (Q = 4), cuyas probabilidades de transición están dadas por:

$P(0_1 \mid 0) = 0.4 \quad P(0_1 \mid 1) = 0.1$

$P(0_2 \mid 0) = 0.3 \quad P(0_2 \mid 1) = 0.2$

$P(1_2 \mid 0) = 0.2 \quad P(1_2 \mid 1) = 0.3$

$P(1_1 \mid 0) = 0.1 \quad P(1_1 \mid 1) = 0.4$

Empleando la ecuación (2.17) y haciendo $c_1 = 1$ y $c_2 = 17.3$ obtenemos la siguiente tabla de métricas de bit:

Tabla 2.1. Métricas de bit para el canal del Ejemplo 8, Cap. 2 [LIN83].

	0_1	0_2	1_2	1_1
0	10	8	5	0
1	0	5	8	10

Ahora bien, para observar la ejecución del algoritmo de Viterbi asumamos que una palabra de código del diagrama de Trellis para el código (3,1,2) de la Figura 2.4 es transmitida por el canal discreto sin memoria **A**, y que la secuencia cuaternaria recibida es: $\mathbf{r} = (1_1 1_2 0_1,\ 1_1 1_1 0_2,\ 1_1 1_1 0_1,\ 1_1 1_1 1_1,\ 0_1 1_2 0_1,\ 1_2 0_2 1_1,\ 1_2 0_1 1_1)$. La aplicación del algoritmo de Viterbi a esta secuencia recibida se muestra en la siguiente Figura.

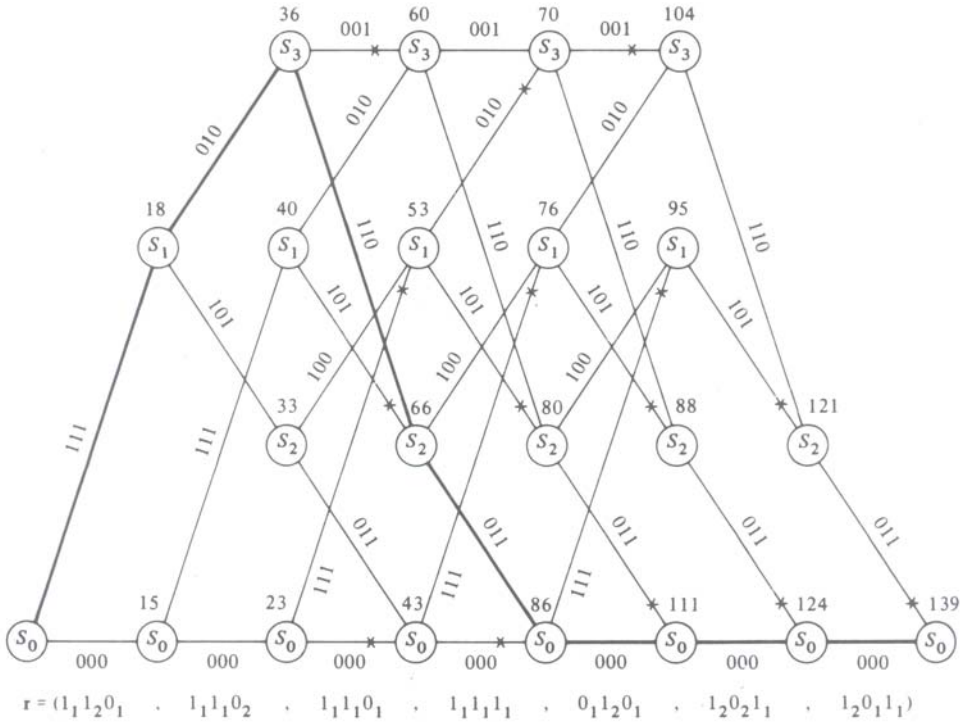

Figura 2.5. Aplicación del algoritmo de Viterbi. [LIN83].

Lo primero que se hace (ver Figura 2.5) es colocar la secuencia recibida bajo el diagrama de Trellis considerado. A continuación se ejecuta el algoritmo de Viterbi paso por paso hasta encontrar la trayectoria de máxima probabilidad en el Trellis. El paso 1 del algoritmo de Viterbi nos indica que calculemos la métrica de la trayectoria que entra a cada estado en la unidad de tiempo $j = 2$. Observando la Figura 2.5, las trayectorias que entran a cada uno de los estados del nivel 2 son:

- al estado S_0: S_0 - S_0 - S_0
- al estado S_1: S_0 - S_0 - S_1
- al estado S_2: S_0 - S_1 - S_2
- al estado S_3: S_0 - S_1 - S_3

Para calcular la métrica de cada una de estas trayectorias observamos que para los primeros dos niveles del Trellis la porción correspondiente de la secuencia recibida es $1_1 1_2 0_1$, $1_1 1_1 0_2$ y empleamos la Tabla 2.1. Para la trayectoria que entra al estado S_0 la métrica se calcula como: $M(1_1 \mid 0)+M(1_2 \mid 0)+M(0_1 \mid 0)+M(1_1 \mid 0)+ M(1_1 \mid 0)+M(0_2 \mid 0)$. De la Tabla 2.1 podemos ver que esta métrica es igual a: 0+5+10+0+0+8 = 23.

De este mismo modo se calculan las métricas restantes y se procede con el algoritmo de Viterbi. En la Figura 2.5, los números de arriba de cada estado representan la métrica de la trayectoria sobreviviente para ese estado, y las trayectorias eliminadas en cada estado se muestran tachadas.

Después de ejecutar el algoritmo de Viterbi se encuentra que para la secuencia recibida **r**, la trayectoria de máxima probabilidad en el Trellis está dada por: S_0 - S_1 - S_3 - S_2 - S_0 - S_0 - S_0 - S_0. Esta trayectoria se encuentra resaltada en negrita en la Figura 2.5. Partiendo del nivel cero y siguiendo esta trayectoria encontramos que el decodificador produce el siguiente estimado **v´**: (1 1 1, 0 1 0, 1 1 0, 0 1 1, 0 0 0, 0 0 0, 0 0 0). Este estimado corresponde a la secuencia de información estimada **u´** = (1 1 0 0 0). Nótese que las *m* ramas finales en cualquier trayecto del Trellis siempre corresponden a entradas 0 y por lo tanto no se consideran parte de la secuencia de información.

De este modo, hemos llevado a cabo un proceso de decodificación convolucional completo empleando el algoritmo de Viterbi.

El método de decodificación de Viterbi es un método óptimo para códigos convolucionales; sin embargo su desempeño depende de la calidad del canal, y el esfuerzo de decodificado crece exponencialmente con la constraint-length de cada código [LIN83].

A pesar de esto, para códigos convolucionales con constraint-length´s bajas la decodificación de Viterbi provee suficientes ganancias de codificación en muchas aplicaciones, y permite que la implementación del decodificador no sea tan compleja. Parte de las ganancias de codificación que pueden lograrse se debe al hecho de que un decodificador de Viterbi puede fácilmente efectuar una decodificación de decisiones suaves, y así no se padece la degradación asociada con una decodificación de decisiones duras [MIC85].

2.3.3. Códigos convolucionales sistemáticos-recursivos (RSC).

Atendiendo a [MIC85], un *código convolucional sistemático* es aquél en que la secuencia de información es contenida, sin ser modificada, en la secuencia codificada (palabra de código) producida por el codificador. De acuerdo con [T.ING98] un codificador convolucional es *recursivo* si existe retroalimentación de los registros de corrimiento del codificador hacia la entrada del mismo.

De este modo, podemos decir que hasta este momento hemos centrado nuestra atención en una clase de códigos convolucionales denominados *no sistemáticos – no recursivos (NSNR)*. Sin embargo, como veremos en el Capítulo 3 de esta Tesis los Turbo-códigos utilizan en su estructura códigos convolucionales que son *sistemáticos* y *recursivos*, lo cual no debe preocuparnos ya que todo lo estudiado hasta aquí sobre codificación convolucional no sistemática – no recursiva aplica de igual manera a codificación convolucional sistemática – recursiva. Por ejemplo, los códigos de este tipo también cuentan con una matriz de función de transferencia, su proceso de codificado puede ser representado por un diagrama de estados y su proceso de decodificado puede efectuarse en la misma forma que se presentó en el Ejemplo 8. Por si fuera poco, los

códigos RSC que se emplean en la estructura de la mayoría de los Turbo-códigos utilizados en una gran cantidad de aplicaciones pueden derivarse a partir de códigos convolucionales NSNR correspondientes en una manera muy simple, como se verá a continuación.

La siguiente Figura nos muestra un codificador convolucional (2, 1, 3) no sistemático – no recursivo y su equivalente versión sistemática – recursiva, que da lugar a un codificador convolucional (2, 1, 3) sistemático – recursivo. En la Figura "X" y "Y" denotan las salidas de los codificadores y "*d*" denota la secuencia de información. Nótese que en el codificador sistemático-recursivo una de sus salidas corresponde directamente a la secuencia de información (sistemático) y que las salidas de las etapas a_1 y a_3 de su registro de corrimiento están retroalimentadas hacia la entrada (recursivo).

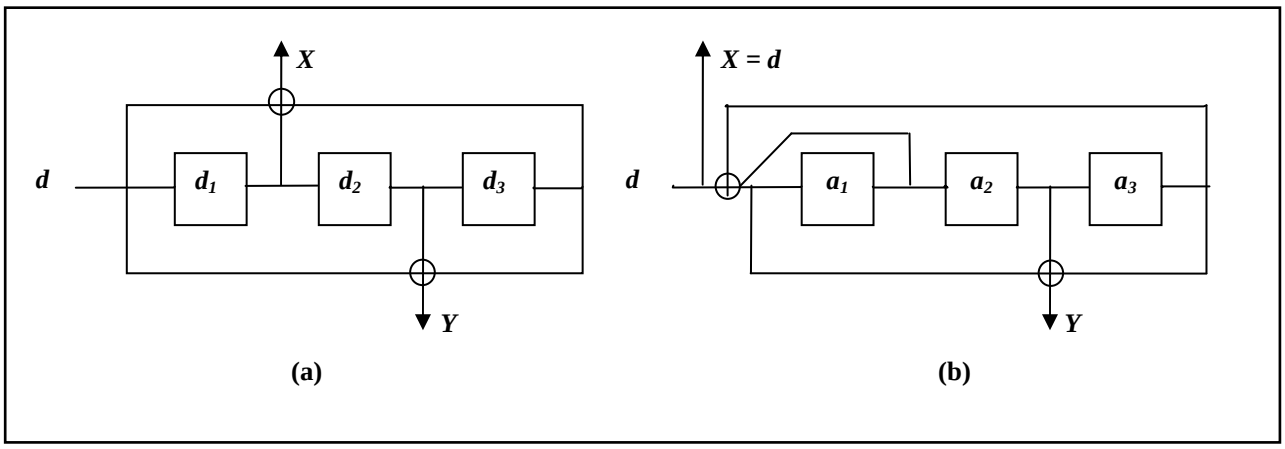

Figura 2.6. Codificador convolucional NSNR de 8 estados **(a)** y su equivalente versión RSC **(b)**. [A.BERR].

A partir de la Figura 2.6 podemos observar que la parte inferior de ambos codificadores es idéntica, y que la parte superior del codificador RSC resulta de conectar su salida X directamente a la entrada y de realimentar hacia esta última las conexiones que producen la secuencia de salida X en el codificador NSNR. Al igual que un codificador NSNR, un codificador RSC puede quedar descrito por medio de una matriz de función de

transferencia, que para un codificador RSC de tasa $r = 1/2$ (como el considerado) toma la forma general:

$$\mathbf{G(D)} = \begin{bmatrix} 1 & \dfrac{\mathbf{g}^{(a)}\mathbf{(D)}}{\mathbf{g}^{(b)}\mathbf{(D)}} \end{bmatrix}$$

donde el "1" representa el polinomio generador asociado a la salida sistemática del codificador RSC y $\mathbf{g}^{(a)}\mathbf{(D)}/\mathbf{g}^{(b)}\mathbf{(D)}$ constituye el polinomio generador asociado a su salida Y. Retomando la Figura 2.6, si $\mathbf{g}^{(1)}\mathbf{(D)}$ constituye el polinomio generador del codificador NSNR asociado a su salida X y $\mathbf{g}^{(2)}\mathbf{(D)}$ es el polinomio generador asociado a su salida Y, entonces $\mathbf{g}^{(a)}\mathbf{(D)} = \mathbf{g}^{(2)}\mathbf{(D)}$ (polinomio generador de la secuencia de salida *no* realimentada en la versión recursiva) y $\mathbf{g}^{(b)}\mathbf{(D)} = \mathbf{g}^{(1)}\mathbf{(D)}$ (polinomio generador de la secuencia de salida realimentada en la versión recursiva). En esta misma forma podemos obtener la versión sistemática – recursiva de cualquier codificador convolucional NSNR de tasa $r = ½$, caracterizado por sus polinomios generadores.

En un codificador sistemático la(s) salida(s) que corresponda(n) directamente a la(s) secuencia(s) de información (de entrada) comunmente se conoce(n) como *secuencia(s) de información*, y la(s) salida(s) restante(s) como *secuencia(s) de paridad*. En la Figura 2.6-(b) se puede observar que este codificador únicamente cuenta con una secuencia de información (X) y una secuencia de paridad (Y).

De acuerdo con [A.BERR], la razón por la que los Turbo-códigos emplean códigos convolucionales sistemáticos-recursivos en su estructura se debe principalmente a que este tipo de códigos ofrece ciertas ventajas importantes con respecto a los códigos convolucionales no sistemáticos-no recursivos. "La primera es conceptual – comenta Berrou – y se debe a que un código convolucional sistemático-recursivo (RSC) está basado en un permutador pseudo-aleatorio, lo cual es bueno si recordamos que Shannon utilizó

códigos aleatorios para calcular el potencial teórico de la codificación del canal. La segunda ventaja es decisiva para tasas de codificación elevadas y/o altos niveles de ruido: un RSC simplemente funciona mejor". (Berrou, 2003: 110).

2.4. Conclusiones del capítulo.

En este Capítulo presentamos las estrategias existentes para el control de errores en los sistemas de comunicación digitales: *Forward Error Correction* (*FEC*) y *Automatic Repeat Request* (*ARQ*). Debido a que en esta Tesis la estrategia de interés la constituye FEC, se realizó una breve descripción de los esquemas ARQ existentes.

El esquema FEC fue el que Claude Shannon consideró era el medio para lograr una transmisión virtualmente libre de errores a una tasa de transferencia igual a la capacidad del canal. A partir de entonces el campo de investigación de la codificación para control de errores siguió dos trayectorias principales: la *codificación de bloque* y la *codificación convolucional*.

Dado que la codificación convolucional representa la base de la Turbo-codificación, en el Capítulo sólo se trataron los conceptos que consideramos más importantes acerca de la codificación de bloque, como la distancia de Hamming y el peso de Hamming que son dos parámetros ampliamente utilizados en la teoría de codificación del canal.

Por la misma razón, la codificación convolucional fue analizada con gran detalle: se estudiaron los métodos de *convolución*, *matriz generadora*, y *matriz de función de transferencia* para obtener la secuencia codificada (palabra de código) producida por un codificador (n, k, m) a partir de una secuencia de información dada; se analizó el *diagrama de estados* y se revisaron las *métricas de distancia* para codificadores convolucionales; y

finalmente se estudió el *algoritmo de Viterbi* como el método para efectuar una decodificación convolucional de máxima probabilidad con ayuda del *diagrama de Trellis*.

Tanto el diagrama de estados como el diagrama de Trellis representan dos poderosas herramientas visuales para analizar de manera sencilla el proceso de codificado de cualquier tipo de codificador convolucional. Los códigos convolucionales sistemáticos-recursivos, expuestos al final del Capítulo, representan una parte fundamental de los Turbo-códigos, y a pesar de que su análisis matemático es más complejo que el de los códigos convolucionales no sistemáticos-no recursivos, los diagramas mencionados podrán emplearse siempre que se requiera de un análisis representativo de dichos códigos.