(19) OFICINA ESPAÑOLA DE PATENTES Y MARCAS

ESPAÑA

(11) Número de publicación: **2 358 551**

(51) Int. Cl.:
H03M 7/40 (2006.01)

(12) TRADUCCIÓN DE PATENTE EUROPEA T3

(96) Número de solicitud europea: **07864208 .9**
(96) Fecha de presentación : **09.11.2007**
(97) Número de publicación de la solicitud: **2092649**
(97) Fecha de publicación de la solicitud: **26.08.2009**

(54) Título: **Codificación eficiente en términos de memoria de códigos de longitud variable.**

(30) Prioridad: **14.11.2006 US 865827 P**
22.11.2006 US 867081 P
17.08.2007 US 840362

(45) Fecha de publicación de la mención BOPI:
11.05.2011

(45) Fecha de la publicación del folleto de la patente:
11.05.2011

(73) Titular/es: **QUALCOMM Incorporated**
Attn: International Ip Administration
5775 Morehouse Drive
San Diego, California 92121, US

(72) Inventor/es: **Reznik, Yuriy**

(74) Agente: **Carpintero López, Mario**

DESCRIPCIÓN

Codificación eficiente en términos de memoria de códigos de longitud variable

Campo técnico

La presente divulgación se refiere a la compresión de datos y, más específicamente, a la compresión de datos usando códigos de longitud variable (VLC).

Antecedentes

La compresión de datos se usa extensamente en una gran variedad de aplicaciones, para reducir el consumo del espacio de almacenamiento de datos, el ancho de banda de la transmisión, o ambos. Ejemplos de aplicaciones de la compresión de datos incluyen la codificación de vídeo, imagen, voz y audio digitales. La codificación de vídeo digital, por ejemplo, se usa en una amplia gama de dispositivos, que incluye televisores digitales, sistemas de difusión directa digital, dispositivos de comunicación inalámbrica, agendas electrónicas (PDA), ordenadores portátiles o de sobremesa, cámaras digitales, dispositivos de grabación digital, dispositivos de juegos de vídeo, radioteléfonos celulares o satelitales, o similares. Los dispositivos de vídeo digital implementan técnicas de compresión de vídeo, tales como MPEG-2, MPEG-4 o Codificación Avanzada de Vídeo (AVC) H.264 / MPEG-4, para transmitir y recibir más eficazmente el vídeo digital.

En general, las técnicas de compresión de vídeo realizan la predicción espacial, la estimación de movimiento y la compensación de movimiento para reducir o suprimir la redundancia inherente en los datos de vídeo. En particular, la intracodificación se apoya en la predicción espacial para reducir o suprimir la redundancia espacial en el vídeo dentro de una trama de vídeo dada. La intercodificación se apoya en la predicción temporal para reducir o suprimir la redundancia temporal en el vídeo dentro de tramas adyacentes. Para la intercodificación, un codificador de vídeo efectúa la estimación de movimiento para rastrear el movimiento de bloques de vídeo coincidentes entre dos o más tramas adyacentes. La estimación de movimiento genera vectores de movimiento, que indican el desplazamiento de bloques de vídeo con respecto a los correspondientes bloques de vídeo en una o más tramas de referencia. La compensación de movimiento usa el vector de movimiento para generar un bloque de vídeo de predicción a partir de una trama de referencia. Después de la compensación de movimiento, se forma un bloque de vídeo residual restando el bloque de vídeo de predicción del bloque de vídeo original.

Un codificador de vídeo aplica procesos de transformación, cuantización y entropía para reducir adicionalmente la tasa de bits del bloque residual producido por el proceso de codificación de vídeo. Las técnicas de codificación de entropía se usan en las etapas finales de un codificador-descodificador de vídeo (CODEC), y en diversas otras aplicaciones de codificación, antes del almacenamiento o la transmisión de los datos codificados. La codificación de entropía implica generalmente la aplicación de códigos aritméticos o códigos de longitud variable (VLC) para comprimir adicionalmente los coeficientes residuales producidos por las operaciones de transformación y cuantización. Los ejemplos de técnicas de codificación de entropía incluyen la codificación aritmética binaria adaptable al contexto (CABAC) y la codificación de longitud variable adaptable al contexto (CAVLC), que pueden usarse como modalidades alternativas de codificación de entropía en algunos codificadores. Un descodificador de vídeo efectúa la descodificación de entropía para descomprimir la información residual para cada uno de los bloques, y reconstruye el vídeo codificado usando información de movimiento y la información residual.

El artículo “A low power variable length decoder for MPEG-2 based on nonuniform fine-grain table partitioning” [“Un descodificador de longitud variable de baja potencia para MPEG-2, basado en la partición no uniforme de tablas de granulado fino”], de Seong Hwan Cho et al, IEEE Transactions on Very Large Scale Integration (VLSI) Systems [Transacciones del IEEE en Sistemas de Integración en Muy Gran Escala (VLSI)], Centro de Servicios del IEEE, Piscataway, NJ, EE UU, Vol. 7, Nº 2, junio de 1999 (1999-06), XP011063421 ISSN: 1063-8210, revela un procedimiento de codificación que incluye generar valores parciales de palabras de código que especifican palabras de código de longitud variable, generar un indicador de omisión y almacenar los valores parciales en una estructura de datos en una memoria.

Resumen

En general, la presente divulgación se orienta a técnicas para la codificación de longitud variable (VLC) adaptable, eficiente en términos de memoria y de baja complejidad, de datos para una gran variedad de aplicaciones, tales como la codificación de datos de vídeo, imagen, audio o voz digitales. Las técnicas pueden aprovechar las propiedades de conjuntos específicos de palabras de código para dar soporte a estructuras de datos muy compactas.

La presente divulgación proporciona un procedimiento que comprende generar valores parciales de palabras de código básicas, correspondiendo las palabras de código básicas a niveles de un árbol de codificación que especifica palabras de código canónicas de longitud variable, estando los valores parciales basados en un

desplazamiento de las palabras de código básicas en un cierto número de bits, generar un indicador de omisión que instruye a un descodificador para omitir un cierto número de bits en un flujo de bits a descodificar antes de avanzar a un nivel seleccionado del árbol de codificación, y almacenar los valores parciales y el indicador de omisión en una estructura de datos en una memoria.

La presente divulgación también proporciona un medio tangible legible por ordenador que incluye un programa de ordenador para implementar el procedimiento.

El procedimiento puede implementarse usando un dispositivo que comprende un procesador configurado para generar los valores parciales de palabras de código básicas y el indicador de omisión, y una memoria que almacena los valores parciales y el indicador de omisión en una estructura de datos.

Un dispositivo de descodificación puede comprender una memoria que almacena una estructura de datos que comprende valores parciales de palabras de código básicas para niveles de un árbol de codificación que especifica palabras de código de longitud variable, y un indicador de omisión que instruye a un descodificador para omitir un cierto número de bits en un flujo de bits, a descodificar antes de avanzar a un nivel seleccionado del árbol de codificación, y un descodificador que accede a la memoria para descodificar una de las palabras de código del flujo de bits, en base a los valores parciales y el indicador de omisión en la estructura de datos almacenada.

En un aspecto adicional, la presente divulgación proporciona un procedimiento de descodificación que comprende acceder a una estructura de datos almacenada en la memoria, en donde la estructura de datos comprende valores parciales de palabras de código básicas para niveles de un árbol de codificación que especifica palabras de código de longitud variable, y un indicador de omisión que instruye a un descodificador para omitir un cierto número de bits en un flujo de bits a descodificar, antes de avanzar a un nivel seleccionado del árbol de codificación, y descodificar una entre las palabras de código del flujo de bits, en base a los valores parciales y al indicador de omisión en la estructura de datos almacenada.

La presente divulgación proporciona adicionalmente un procedimiento adicional que comprende realizar la codificación de longitud variable según una estructura de códigos, en donde la estructura de códigos define grupos de palabras de código en un árbol de codificación, cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos y las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, en donde el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud, y generar un resultado de la codificación de longitud variable para al menos uno entre el almacenamiento en memoria, la transmisión a un dispositivo o la presentación a un usuario.

Puede proporcionarse adicionalmente un medio tangible legible por ordenador que comprende instrucciones para causar que un procesador efectúe la codificación de longitud variable según una estructura de códigos, en donde la estructura de códigos define grupos de palabras de código en un árbol de codificación, cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos y las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, en donde el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud, y genere un resultado de la codificación de longitud variable para el menos uno entre el almacenamiento en memoria, la transmisión a un dispositivo o la presentación a un usuario.

La presente divulgación también proporciona un dispositivo que comprende un procesador, adicionalmente configurado para realizar la codificación de longitud variable según una estructura de códigos, en donde la estructura de códigos define grupos de palabras de código en un árbol de codificación, cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos y las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, y en donde el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud, y generar un resultado de la codificación de longitud variable para al menos uno entre el almacenamiento en memoria, la transmisión a un dispositivo o la presentación a un usuario.

El procedimiento puede comprender adicionalmente, para una estructura de códigos que define grupos de palabras de código en un árbol de codificación que especifica palabras de código de longitud variable, en donde cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos, y las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, en

donde el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud, efectuando la codificación de longitud variable usando palabras de código básicas para cada uno de los subgrupos, posiciones de palabras de código dentro de cada uno de los grupos, un cierto número de palabras de código dentro de cada uno de los primeros subgrupos y longitudes de las palabras de código dentro de cada uno de los subgrupos, y generando un resultado de la codificación de longitud variable para al menos uno entre el almacenamiento en memoria, la transmisión a un dispositivo o la presentación a un usuario.

La presente divulgación también proporciona un dispositivo que comprende adicionalmente, para una estructura de código, medios para definir grupos de palabras de código en un árbol de codificación que especifica palabras de código de longitud variable, en donde cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos y las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, en donde el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud, medios para realizar la codificación de longitud variable usando palabras de código básicas para cada uno de los subgrupos, posiciones de palabras de código dentro de cada uno de los grupos, un cierto número de palabras de código dentro de cada uno de los primeros subgrupos y longitudes de las palabras de código dentro de cada uno de los subgrupos, y medios para generar un resultado de la codificación de longitud variable para al menos uno entre el almacenamiento en memoria, la transmisión a un dispositivo o la presentación a un usuario.

Un medio tangible legible por ordenador puede comprender adicionalmente instrucciones para causar que un procesador realice, para una estructura de códigos que define grupos de palabras de código en un árbol de codificación que especifica palabras de código de longitud variable, en donde cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos y las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, en donde el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud, la codificación de longitud variable usando palabras de código básicas para cada uno de los subgrupos, posiciones de palabras de código dentro de cada uno de los grupos, un cierto número de palabras de código dentro de cada uno de los primeros subgrupos y longitudes de las palabras de código dentro de cada uno de los subgrupos, y generar un resultado de la codificación de longitud variable para al menos uno entre el almacenamiento en memoria, la transmisión a un dispositivo o la presentación a un usuario.

La presente divulgación proporciona adicionalmente un dispositivo que comprende, para una estructura de códigos que define grupos de palabras de código en un árbol de codificación que especifica palabras de código de longitud variable, en donde cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos y las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, en donde el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud, un procesador configurado para realizar la codificación de longitud variable usando palabras de código básicas para cada uno de los subgrupos, posiciones de palabras de código dentro de cada uno de los grupos, un cierto número de palabras de código dentro de cada uno de los primeros subgrupos, y longitudes de las palabras de código dentro de cada uno de los subgrupos, y generar un resultado de la codificación de longitud variable para al menos uno entre el almacenamiento en memoria, la transmisión a un dispositivo o la presentación a un usuario.

La presente divulgación proporciona adicionalmente un medio tangible legible por ordenador que comprende una estructura de datos para la codificación de longitud variable que usa una estructura de códigos de longitud variable que define grupos de palabras de código en un árbol de codificación, cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos y las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, en donde el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud distinta a la primera longitud.

La presente divulgación proporciona adicionalmente un dispositivo de circuito integrado que comprende una memoria que almacena una estructura de datos que comprende valores parciales de palabras de código básicas para niveles de un árbol de codificación que especifica palabras de código de longitud variable, y un indicador de omisión que instruye a un descodificador para omitir un cierto número de bits en un flujo de bits a descodificar, antes de avanzar a un nivel seleccionado del árbol de codificación, y un descodificador que accede a la memoria para descodificar una de las palabras de código del flujo de bits, en base a los valores parciales y al indicador de

omisión en la estructura de datos almacenada.

La presente divulgación proporciona adicionalmente un dispositivo de mano para comunicación inalámbrica que comprende una memoria que almacena una estructura de datos que comprende valores parciales de palabras de código básicas para niveles de un árbol de codificación que especifica palabras de código de longitud variable, y un indicador de omisión que instruye a un descodificador para omitir un cierto número de bits en un flujo de bits a descodificar, antes de avanzar a un cierto nivel del árbol de codificación, un descodificador que accede a la memoria para descodificar una de las palabras de código del flujo de bits, en base a los valores parciales y al indicador de omisión en la estructura de datos almacenada, un receptor para recibir las palabras de código desde un codificador por comunicación inalámbrica y un dispositivo de salida que presenta la salida a un usuario en base, al menos en parte, a las palabras de código descodificadas.

La presente divulgación proporciona adicionalmente un dispositivo de circuitos integrados que comprende un procesador configurado para realizar la codificación de longitud variable según una estructura de código, en donde la estructura de código define grupos de palabras de código en un árbol de codificación, cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos y las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, en donde el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud, y en donde el procesador está configurado para generar un resultado de la codificación de longitud variable para al menos uno entre el almacenamiento en memoria, la transmisión a un dispositivo o la presentación a un usuario.

La presente divulgación proporciona adicionalmente un dispositivo de mano de comunicación inalámbrica que comprende un procesador configurado para realizar la codificación de longitud variable, según una estructura de códigos, en donde la estructura de códigos define grupos de palabras de código en un árbol de codificación, cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos y las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, en donde el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud, y en donde el procesador está configurado para generar un resultado de la codificación de longitud variable para al menos uno entre el almacenamiento en la memoria, la transmisión a un dispositivo o la presentación a un usuario.

Las técnicas descritas en la presente divulgación pueden implementarse en hardware, software, firmware o cualquier combinación de los mismos. Si se implementan en software, el software puede ejecutarse en uno o más procesadores, tales como un microprocesador, un circuito integrado específico de la aplicación (ASIC), una formación de compuertas programables en el terreno (FPGA), o un procesador de señales digitales (DSP), u otro circuito lógico equivalente, integrado o discreto. El software que ejecuta las técnicas puede almacenarse inicialmente en un medio legible por ordenador y cargarse y ser ejecutado por un procesador. En consecuencia, la presente divulgación también contempla productos de programa de ordenador que comprenden un medio legible por ordenador que comprende instrucciones para causar que un procesador lleve a cabo cualquiera entre una cierta variedad de técnicas, según se describe en la presente divulgación.

Los detalles de uno o más aspectos de la presente divulgación se exponen en los dibujos adjuntos y la descripción más adelante. Otras características, objetos y ventajas de las técnicas descritas en la presente divulgación serán evidentes a partir de la descripción y los dibujos, y a partir de las reivindicaciones.

Breve descripción de los dibujos

La FIG. 1 es un diagrama en bloques que ilustra un sistema de codificación y descodificación de vídeo.

La FIG. 2 es un diagrama en bloques que ilustra un ejemplo de un codificador de vídeo.

La FIG. 3 es un diagrama en bloques que ilustra un ejemplo de un descodificador de vídeo.

La FIG. 4 es un diagrama que ilustra un ejemplo de un árbol binario de codificación.

La FIG. 5 es un gráfico que ilustra la tasa de redundancia de un código de bloques adaptable con un comportamiento asintótico.

La FIG. 6 es un diagrama de un árbol binario que ilustra grupos de bloques, subgrupos y palabras de código básicas.

Las FIGS. 7A y 7B son gráficos que ilustran la comparación de tasas de redundancia de un código adaptable de

bloques con distintos valores de p.

La FIG. 8 es un gráfico que ilustra la sensibilidad de la redundancia a la asimetría de los datos de origen.

La FIG. 9 es un diagrama de flujo que ilustra un procedimiento para construir la codificación de longitud variable, eficaz en términos de memoria, para distribuciones monótonas, según un aspecto de la presente divulgación.

La FIG. 10 es un diagrama de flujo que ilustra un procedimiento para codificar símbolos, usando códigos de longitud variable construidos según el procedimiento de la FIG. 9.

La FIG. 11 es un diagrama de flujo que ilustra un procedimiento para descodificar códigos de longitud variable, construidos según el procedimiento de la FIG. 9.

La FIG. 12 es un diagrama de flujo que ilustra un procedimiento para construir códigos adaptables de bloques según otro aspecto de la presente divulgación.

La FIG. 13 es un diagrama de flujo que ilustra un procedimiento para codificar bloques usando códigos de longitud variable, construidos según el procedimiento de la FIG. 12.

La FIG. 14 es un diagrama en bloques que ilustra un procedimiento para descodificar códigos de longitud variable, construidos según el procedimiento de la FIG. 12.

Descripción detallada

En general, la presente divulgación está orientada a técnicas para la codificación de longitud variable (VLC) adaptable, eficiente en términos de memoria y de baja complejidad, de los datos para una gran variedad de aplicaciones, tales como la codificación de datos de vídeo, imagen, audio o voz digitales. Las técnicas pueden explotar las propiedades de conjuntos específicos de palabras de código para dar soporte a estructuras de datos muy compactas. Las técnicas también pueden dar soporte a la codificación y descodificación adaptable de baja complejidad de secuencias binarias producidas por fuentes sin memoria. Aunque las técnicas descritas en la presente divulgación pueden ser aplicables a una amplia variedad de aplicaciones prácticas, incluyen la compresión y codificación general de datos, la presente divulgación se referirá a la codificación y descodificación de vídeo digital con fines de ejemplo e ilustración.

Para dar soporte a estructuras de datos compactas, las técnicas reveladas no necesitan apoyarse en ningún esquema específico de construcción de código. Por ejemplo, pueden usarse los esquemas de Huffman, Shannon, Shannon-Fano, Gilbert-Moore u otros esquemas de construcción de código. Sin embargo, se supone que tal código se construye para un origen con probabilidades monótonamente crecientes de símbolos a partir de un alfabeto de símbolos de entrada. Más específicamente, se supone que las palabras de código tienen longitudes monótonamente decrecientes, o al menos longitudes no crecientes, y que las palabras de código de la misma longitud tienen el mismo orden lexicográfico que los símbolos en el alfabeto de entrada que representan.

El orden lexicográfico deseado puede lograrse reordenando el alfabeto de entrada. Para tales palabras de código, los valores de palabras de código básicas pueden calcularse para cada nivel de un árbol de codificación. Los valores de palabras de código básicas representan las palabras de código canónicas lexicográficamente más pequeñas en cada nivel del árbol de codificación. Los valores de palabras de código básicas y los índices de sus respectivos símbolos pueden almacenarse en una formación reordenada. Los índices pueden almacenarse como valores de desplazamiento para cada nivel poblado en el árbol. Un proceso de descodificación puede implicar la comparación de un almacén temporal del flujo de bits con valores de palabras de código básicas alineadas a izquierda, seguida por el cálculo directo sencillo de un índice de un símbolo descodificado.

Las propiedades anteriores pueden usarse para describir unívocamente un tal código, con una estructura de datos que pueda comprimirse adicionalmente para producir una estructura de datos muy compacta que facilite la descodificación incremental de códigos de longitud variable. Por ejemplo, los valores de palabras de código básicas alineadas a izquierda tendrán habitualmente grandes cantidades de ceros iniciales de derecha a izquierda. Los números de ceros iniciales son monótonamente decrecientes según el proceso avanza hacia capas más profundas en un árbol de codificación aplicable. Por tanto, cuando los bits se descodifican secuencialmente a partir de la primerísima capa y avanzando hacia abajo, algunos de los bits cero iniciales pueden omitirse sin afectar la precisión del proceso de descodificación.

Los ceros iniciales pueden quitarse en incrementos fijos, p. ej., incrementos de 8 bits, lo que es conveniente para la gestión de almacenes temporales de flujos de bits en ordenadores convencionales de 8 bits por octeto. Una tabla adicional de uno o más indicadores de omisión puede proporcionarse para gestionar este proceso. Como ejemplo, un indicador de omisión dirige a un descodificador para desplazarse hacia adelante, en un almacén temporal de flujos de bits, en un incremento fijo, de modo que los valores distintos de palabras de código básicas puedan

distinguirse cuando se eliminan los ceros iniciales. Con la eliminación de los ceros iniciales, el ancho de la formación resultante de valores modificados de palabras de código básicas puede reducirse significativamente, conservando por ello la utilización de la memoria.

Por tanto, la presente divulgación contempla un procedimiento que comprende generar valores parciales de palabras de código básicas para niveles de un árbol de codificación que especifica palabras de código de longitud variable, generar un indicador de omisión que instruye a un descodificador para omitir un cierto número de bits en un flujo de bits a descodificar antes de avanzar a un nivel seleccionado del árbol de codificación, y almacenar los valores parciales y el indicador de omisión en una estructura de datos en una memoria. La estructura de datos puede ser cualquiera entre una amplia variedad de estructuras de datos, tales como tablas, listas enlazadas, árboles binarios, árboles radicales, ficheros planos o similares, y puede almacenarse en cualquiera entre una amplia variedad de distintos dispositivos de memoria, tales como las muchas formas de la memoria de acceso aleatorio (RAM), la memoria de sólo lectura (ROM), o ambas. La estructura de datos puede almacenarse en tal memoria dentro de un codificador o un descodificador. Por ejemplo, un descodificador puede acceder a la estructura de datos, o partes del contenido de la estructura de datos, a partir de la memoria asociada al descodificador, para realizar la descodificación de longitud variable de palabras de código de manera eficiente en términos de memoria.

La presente divulgación contempla adicionalmente un procesador configurado para generar valores parciales de palabras de código básicas para niveles de un árbol de codificación que especifica palabras de código de longitud variable, y generar un indicador de omisión que instruye a un descodificador para omitir un cierto número de bits en un flujo de bits a descodificar, antes de avanzar a un nivel seleccionado del árbol de codificación, así como una memoria que almacena los valores parciales y el indicador de omisión en una estructura de datos. Tal información puede almacenarse en una única estructura de datos o en múltiples estructuras de datos. Por consiguiente, la referencia a una estructura de datos puede incluir una o más estructuras de datos que almacenan la información contemplada en la presente divulgación. Un procesador configurado para acceder a una estructura de datos a fin de realizar la codificación de longitud variable puede implementarse dentro de un dispositivo de origen o un dispositivo receptor, o dentro de un dispositivo distinto que genera estructuras de datos que definen estructuras de códigos para su uso por parte de un codificador y / o descodificador al llevar a cabo la codificación de longitud variable.

Según esta técnica para lograr estructuras de datos compactas, cada longitud de palabra de código válida puede corresponder a un nivel con un nodo externo en un árbol de códigos. Como se ha expuesto anteriormente, la estructura de datos puede incluir valores parciales de palabras de código básicas y uno o más indicadores de omisión. Más específicamente, en algunas realizaciones, la estructura de datos puede contener, para cada longitud de palabra de código válida, la siguiente información: (a) un valor parcial de la palabra de código lexicográficamente más pequeña (o más grande) en el nivel actual del árbol de códigos, (b) el número de bits en el valor parcial, (c) un valor del símbolo correspondiente a la palabra de código lexicográficamente más pequeña (o más grande) y (d) un indicador que instruye al descodificador para omitir un cierto número de bits antes de avanzar al siguiente nivel del árbol de códigos. En consecuencia, la estructura de datos puede incluir adicionalmente valores para símbolos representados por las palabras de código básicas y las longitudes de valores parciales de las palabras de código básicas, es decir, el número de bits en cada valor parcial de una palabra de código básica. Las técnicas de codificación y de descodificación pueden usar esta estructura de datos para identificar el nivel correspondiente a una palabra de código a producir o descodificar, y calcular luego directamente el valor de la palabra de código o del símbolo descodificado. En consecuencia, la estructura de datos puede almacenarse en la memoria de un codificador o descodificador de vídeo, codificador o descodificador de imagen, codificador o descodificador de audio o codificador o descodificador de voz, algunos de los cuales pueden construirse como CODEC combinados.

Los ejemplos de técnicas existentes para codificar o descodificar códigos de longitud variable se describen en los documentos de A. Moffat y A. Turpin, “On the Implementation of Minimum-Redundancy Prefix Codes” [“Sobre la implementación de códigos de prefijos de redundancia mínima”], IEEE Trans. Communications, 45 (10) (1997) 1200-1207, de J. B. Connell, “A Huffman-Shannon-Fano Code” [“Un código de Huffman-Shannon-Fano”], Proc. IEEE, Julio de 1973, 1046-1047 y de A. Brodnik y S. Carlsson, “Sublinear Decoding of Huffman Codes Almost in Place” [“Descodificación sublineal de códigos de Huffman casi en su sitio”], Taller DIMACS sobre Códigos y Árboles: Enfoques Teóricos Algorítmicos y de información, Octubre de 1998, Universidad Rutgers, Centro DIMACS, NJ. En comparación con estas técnicas existentes, las técnicas reveladas para lograr estructuras de datos compactas pueden ofrecer ciertas ventajas.

Como un primer ejemplo, una estructura de datos producida por la técnica revelada puede usar una cantidad mucho más pequeña de memoria, debido al hecho de que sólo se almacenan y usan valores parciales de las palabras de código lexicográficamente más pequeñas, p. ej., por parte de un descodificador de vídeo. Como otro ejemplo, la técnica revelada puede usar el acceso incremental a los datos del flujo de bits, permitiendo que un almacén temporal del flujo de bits esté presentado por un registro razonablemente corto, y actualizado cada cualquier intervalo conveniente, p. ej., mediante una indicación de omisión, rebajando adicionalmente la complejidad de la

implementación.

Por ejemplo, en algunas implementaciones, un registro de 32 bits puede ser suficiente incluso para códigos muy largos. Además, las actualizaciones pueden hacerse a intervalos de 8 bits. Globalmente, la técnica revelada puede reducir significativamente la complejidad de la representación y la codificación / descodificación de códigos de longitud variable, lo que puede permitir a los diseñadores de algoritmos de compresión utilizar libros de códigos mucho más grandes y, por ello, más eficientes.

Además, para dar soporte a la codificación y descodificación adaptable, de baja complejidad, de secuencias binarias producidas por fuentes sin memoria, las técnicas reveladas pueden implementar códigos universales de bloque construidos para un conjunto de contextos identificados por los números de bits no nulos en bits anteriores en una secuencia. Estas técnicas para la codificación y descodificación adaptable, de baja complejidad, pueden apoyarse en una fórmula revelada para la redundancia asintótica de tales códigos, lo que refina la estimación descrita en el documento de R. E. Krichevskiy y V. K. Trofimov, “The Performance of Universal Encoding” [“Las prestaciones de la codificación universal”], IEEE Trans. Information Theory, 27 (1981) 199-207.

Estas técnicas pueden usar una formación de códigos de Huffman diseñados para varias densidades estimadas, e indizados por los números de bits no nulos en bloques (contextos) anteriores en una secuencia. Utilizando incluso bloques de bits de tamaño modesto n = 8 ... 16 (y usando unos correspondientes 1,5 ... 5k octetos de memoria), tales técnicas pueden lograr unas prestaciones de compresión comparables o superiores a las de otros algoritmos existentes, tales como el algoritmo codificador-Q descrito en el documento de W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, Jr. y R. B. Arps, “An overview of the basic principles of the Q-Coder adaptive binary arithmetic coder” [“Un panorama de los principios básicos del codificador aritmético binario adaptable Codificador-Q”], IBM J. Res. Dev., 32 (6) (1988) 717, que se usa en el estándar de codificación de imágenes JBIG, o el algoritmo CABAC descrito en el documento de D. Marpe, H. Schwarts y T. Wiegand, “Context-Based Adaptative Binary Arithmetic Coding in the H.264/AVC video compression standard” [“Codificación aritmética binaria adaptable basada en el contexto en el estándar de compresión de vídeo H.264 / AVC”], IEEE Trans. on CSVT, 13(7):620 636, julio de 2003, que se usa en los estándares ITU-T H.264 / MPEG AVC para la compresión de vídeo.

La codificación y descodificación adaptable de baja complejidad puede basarse en la conclusión de que, en un modelo sin memoria, la probabilidad de un bloque de bits, o su estimación, depende sólo de su peso (el número de bits no nulos) y no de un patrón efectivo de sus bits. Por tanto, un conjunto de todos los bloques posibles de alguna longitud fija puede dividirse en distintos grupos que contienen bloques del mismo peso (y, en consecuencia, la misma probabilidad). Puede suponerse que cada grupo almacena bloques en un orden lexicográfico, bien naturalmente o por reordenamiento.

Es una propiedad conocida de los códigos de redundancia mínima (tales como los códigos de Huffman o Shannon) que cada grupo de bloques equiprobables puede incluir a lo sumo dos subgrupos, donde los bloques en cada tal subgrupo están codificados por palabras de código de la misma longitud. Sin restricción sobre la generalidad, puede suponerse adicionalmente que la longitud de las palabras de código en el primer subgrupo es más corta que la longitud de las palabras de código en el segundo subgrupo. Debido a que los bloques (o palabras) dentro de un grupo siguen el orden lexicográfico, puede dividirse sencillamente entre un subgrupo con mayor longitud de palabras de código y un subgrupo con una menor longitud de palabras de código. Un valor de índice indica la posición del bloque dentro de un grupo. Al bloque (o palabra) lexicográficamente más pequeño en cada subgrupo se asigna una palabra de código básica. Dada una palabra de código básica, las palabras de código restantes en cada subgrupo pueden calcularse inmediatamente.

Por tanto, la codificación de longitud variable puede ser realizada, p. ej., por un codificador o descodificador usando una estructura de códigos que define grupos de bloques o palabras de entrada y sus respectivas palabras de código de salida en un árbol de codificación, en el cual cada uno de los grupos incluye palabras de código que representan bloques (o palabras) con los mismos pesos, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos, en el cual el primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud. Los bloques en cada uno de los grupos se ordenan lexicográficamente y se dividen luego en subgrupos, de modo que el orden lexicográfico se retenga en cada uno de los subgrupos. Además, las palabras de código correspondientes a cada bloque dentro de un subgrupo se asignan de modo tal que también sigan el mismo orden lexicográfico y faciliten la codificación y descodificación por cálculo directo.

En base a esta disposición de bloques y sus grupos y subgrupos, las palabras de código pueden calcularse directamente usando un proceso simplificado. Por ejemplo, al obtener un peso y un valor de índice para un bloque, si el valor del índice es menor que el número máximo de bloques en el primer subgrupo, entonces se selecciona un primer subgrupo para localizar la palabra de código. En caso contrario, se selecciona el segundo subgrupo para localizar la palabra de código. Luego, al capturar la palabra de código básica para el subgrupo seleccionado, la palabra de código se calcula inmediatamente sumando el valor de la palabra de código básica a un valor

determinado en base al valor de índice del bloque dentro del subgrupo seleccionado. Para los fines de este segundo aspecto general de la presente divulgación, los términos “bloques” o “palabras” pueden usarse intercambiablemente para referirse en general a cantidades de entrada a codificar según un esquema de codificación. Un bloque o palabra puede incluir una secuencia de símbolos, que pueden formarse a partir de un alfabeto de entrada, tal como un alfabeto binario {0, 1}. Las palabras de código, por lo general, se refieren a cantidades de salida asignadas a bloques (o palabras) como resultado del esquema de codificación.

Estos, y otros, aspectos de las técnicas descritas en la presente divulgación se describirán más adelante en mayor detalle. Las técnicas pueden usarse juntas o independientemente, y pueden implementarse en cualquiera entre una gran variedad de sistemas y dispositivos de codificación y descodificación, incluyendo sistemas y dispositivos configurados para la codificación o descodificación de datos de vídeo, imagen, audio o voz digitales. Con fines de ejemplo e ilustración, la presente divulgación describirá la aplicación de tales técnicas a la codificación y descodificación de vídeo digital, sin limitación en cuanto a la aplicación práctica general de la compresión y codificación de datos, u otras aplicaciones específicas a distintos tipos de datos.

La FIG. 1 es un diagrama en bloques que ilustra un sistema 10 de codificación y descodificación de vídeo. Como se muestra en la FIG. 1, el sistema 10 incluye un dispositivo 12 de origen que transmite vídeo codificado a un dispositivo receptor 14 mediante un canal 16 de comunicación. El dispositivo 12 de origen puede incluir una fuente 18 de vídeo, un codificador 20 de vídeo y un transmisor 22. El dispositivo receptor 14 puede incluir un receptor 24, un descodificador 26 de vídeo y un dispositivo 28 de visualización de vídeo. El sistema 10 puede configurarse para aplicar técnicas para la codificación de longitud variable (VLC) adaptable, eficiente en términos de memoria y de baja complejidad, de datos de vídeo digital. En particular, las técnicas de VLC adaptable, eficiente en términos de memoria y de baja complejidad, pueden usarse para la codificación de entropía de los coeficientes de bloques residuales producidos por un proceso de codificación predictiva de vídeo. Las técnicas pueden aplicarse a esquemas de codificación de vídeo que codifican las posiciones de coeficientes de transformación no nulos usando ristras de ceros, o a otros esquemas de codificación de vídeo.

En el ejemplo de la FIG. 1, el canal 16 de comunicación puede comprender cualquier medio de comunicación inalámbrico o cableado, tal como un espectro de frecuencia de radio (RF) o una o más líneas de transmisión física, o cualquier combinación de medios inalámbricos y cableados. El canal 16 puede formar parte de una red basada en paquetes, tal como una red de área local, una red de área amplia, o una red global tal como Internet. El canal 16 de comunicación representa generalmente a cualquier medio de comunicación adecuado, o colección de distintos medios de comunicación, para transmitir datos de vídeo desde el dispositivo 12 de origen al dispositivo receptor 14.

El dispositivo 12 de origen genera vídeo para su transmisión al dispositivo 14 de destino. En algunos casos, sin embargo, los dispositivos 12, 14 pueden funcionar de manera esencialmente simétrica. Por ejemplo, cada uno de los dispositivos 12, 14 puede incluir componentes de codificación y descodificación de vídeo. Por tanto, el sistema 10 puede dar soporte a la transmisión de vídeo unidireccional o bidireccional entre los dispositivos 12, 14 de vídeo, p. ej., para el flujo de vídeo, la difusión de vídeo o la telefonía de vídeo. Para otras aplicaciones de compresión y codificación de datos, los dispositivos 12, 14 podrían configurarse para enviar y recibir, o intercambiar, otros tipos de datos, tales como datos de imagen, voz o audio, o combinaciones de dos o más entre datos de vídeo, imagen, voz y audio. En consecuencia, la exposición de las aplicaciones de vídeo se proporciona con fines de ilustración y no debería considerarse como limitadora de los diversos aspectos de la presente divulgación según se describen a grandes rasgos en el presente documento.

El origen 18 de vídeo puede incluir un dispositivo de captura de vídeo, tal como una o más cámaras de vídeo, un archivo de vídeo que contiene vídeo previamente capturado, o una señal de vídeo en vivo desde un proveedor de contenidos de vídeo. Como alternativa adicional, el origen 18 de vídeo puede generar datos basados en gráficos de ordenador, tales como el vídeo de origen, o una combinación de vídeo en vivo y vídeo generado por ordenador. En algunos casos, si el origen 18 de vídeo es una cámara, el dispositivo 12 de origen y el dispositivo receptor 14 pueden formar los llamados fonocámaras o fonovídeos. Por tanto, en algunos aspectos, el dispositivo 12 de origen, el dispositivo receptor 14, o ambos, pueden formar un dispositivo de mano de comunicación inalámbrica, tal como un teléfono móvil. En cada caso, el vídeo capturado, precapturado o generado por ordenador puede ser codificado por el codificador 20 de vídeo para su transmisión desde el dispositivo 12 de origen al descodificador 26 de vídeo del dispositivo receptor 14 de vídeo, mediante el transmisor 22, el canal 16 y el receptor 24. El dispositivo 28 de visualización puede incluir cualquiera entre una gran variedad de dispositivos de visualización, tal como un visor de cristal líquido (LCD), un visor de plasma o un visor orgánico de diodo emisor de luz (OLED).

El codificador 20 de vídeo y el descodificador 26 de vídeo pueden configurarse para dar soporte a la codificación escalable de vídeo para lograr escalabilidad espacial, temporal y / o de razón entre señal y ruido (SNR). En algunos aspectos, el codificador 20 de vídeo y el descodificador 22 de vídeo pueden configurarse para dar soporte a la codificación de escalabilidad de SNR de granularidad fina (FGS) para la SVC (Codificación Escalable de Vídeo). El codificador 20 y el descodificador 26 pueden dar soporte a diversos grados de escalabilidad brindando soporte a la

codificación, transmisión y descodificación de una capa básica y una o más capas escalables de realce. Para la codificación escalable de vídeo, una capa básica lleva datos de vídeo con un nivel mínimo de calidad. Una o más capas de realce llevan un flujo de bits adicional para dar soporte a niveles espaciales, temporales y / o de SNR superiores.

El codificador 20 de vídeo y el descodificador 26 de vídeo pueden funcionar según un estándar de compresión de vídeo, tal como MPEG-2, MPEG-4, ITU-T H.263 o ITU-T H.264 / MPEG-4 Codificación Avanzada de Vídeo (AVC). Aunque no se muestra en la FIG. 1, en algunos aspectos, el codificador 20 de vídeo y el descodificador 26 de vídeo pueden integrarse con un codificador y descodificador de audio, respectivamente, e incluir las unidades adecuadas de MUX-DEMUX, u otro hardware y software, para gestionar la codificación tanto de audio como de vídeo en un flujo común de datos o en flujos de datos distintos. Si es aplicable, las unidades MUX-DEMUX pueden ser conformes al protocolo multiplexador ITU H.223, u otros protocolos tales como el protocolo de datagramas de usuario (UDP).

El estándar H.264 / MPEG-4 (AVC) fue formulado por el Grupo de Expertos en Codificación de Vídeo (VCEG) de la ITU-T (Unión Internacional de Telecomunicaciones - Telecomunicación), junto con el Grupo de Expertos en Películas (MPEG) de ISO / IEC (Organización Internacional de Estándares / Comisión Electrotécnica Internacional), como el producto de una asociación colectiva conocida como el Equipo Conjunto de Vídeo (JVT). El estándar H.264 se describe en la Recomendación H.264 de la ITU-T, “Advanced video coding for generic audiovisual services” [“Codificación avanzada de vídeo para servicios audiovisuales genéricos”], del Grupo de Estudio de la ITU-T, y con fecha de 03 / 2005, que puede mencionarse en el presente documento como el estándar H.264 o la especificación H.264, o el estándar o especificación H.264 / AVC.

El Equipo Conjunto de Vídeo (JVT) continúa trabajando en una extensión de la codificación escalable de vídeo (SVC) al estándar H.264 / MPEG-4 AVC. La especificación de la extensión de la SVC en evolución tiene la forma de un Borrador Conjunto (JD). El Modelo de Vídeo Escalable Conjunto (JSVM) creado por el JVT implementa herramientas para su uso en el vídeo escalable, que pueden usarse dentro del sistema 10 para diversas tareas de codificación descritas en la presente divulgación. Información detallada referida a la codificación de Escalabilidad de SNR de Granularidad Fina (FGS) puede hallarse en los documentos de Borrador Conjunto, p. ej., en el Borrador Conjunto 6 (SVC JD6), de Thomas Wiegand, Gary Sullivan, Julien Reichel, Heiko Schwarz y Mathias Wien, “Joint Draft 6: Scalable Video Coding” [“Borrador Conjunto 6: Codificación Escalable de Vídeo”], JVT-S 201, abril de 2006, Ginebra, y en el Borrador Conjunto 9 (SVC JD9), de Thomas Wiegand, Gary Sullivan, Julien Reichel, Heiko Schwarz y Mathias Wien, “Joint Draft 9 of SVC Amendment” [“Borrador Conjunto 9 de Enmienda de SVC”], JVT-V 201, enero de 2007, Marrakech, Marruecos.

En algunos aspectos, para la difusión de vídeo, las técnicas descritas en la presente divulgación pueden aplicarse a la codificación de vídeo H.264 Realzada para prestar servicios de vídeo en tiempo real en sistemas de multidifusión de multimedios móviles terrestres (TM3) usando la Especificación de Interfaz Aérea Sólo de Enlace Directo (FLO) “Forward Link Only Air Interface Specification for Terrestrial Mobile Multimedia Multicast” [“Especificación de interfaz aérea sólo de enlace directo para multidifusión de multimedios móviles terrestres”], a publicarse como el Estándar Técnico TIA-1099 (la “Especificación FLO”), p. ej., mediante un servidor inalámbrico de difusión de vídeo o dispositivo de mano de comunicación inalámbrica. La Especificación FLO incluye ejemplos que definen la sintaxis y la semántica del flujo de bits y los procesos de descodificación adecuados para la Interfaz Aérea FLO. Alternativamente, el vídeo puede ser difundido según otros estándares tales como DVB-H (difusión de vídeo digital - de mano), ISDB-T (difusión digital de servicios integrados – terrestre) o DMB (difusión de medios digitales). Por tanto, el dispositivo 12 de origen puede ser un terminal inalámbrico móvil, un servidor de flujos de vídeo o un servidor de difusión de vídeo. Sin embargo, las técnicas descritas en la presente divulgación no se limitan a ningún tipo específico de sistema de difusión, multidifusión o de punto a punto. En el caso de la difusión, el dispositivo 12 de origen puede difundir varios canales de datos de vídeo a múltiples dispositivos receptores, cada uno de los cuales puede ser similar al dispositivo receptor 14 de la FIG. 1.

Cada uno entre el codificador 20 de vídeo y el descodificador 26 de vídeo puede implementarse como uno o más microprocesadores, procesadores de señales digitales (DSP), circuitos integrados específicos para la aplicación (ASIC), formaciones de compuertas programables en el terreno (FPGA), lógica discreta, software, hardware, firmware o combinaciones cualesquiera de los mismos. Por tanto, cada uno entre el codificador 20 de vídeo y el descodificador 26 de vídeo puede implementarse, al menos parcialmente, como un chip o dispositivo de circuitos integrados (IC), e incluirse en uno o más codificadores o descodificadores, cada uno de los cuales puede integrarse como parte de un codificador / descodificador combinado (CODEC) en un respectivo dispositivo móvil, dispositivo de abonado, dispositivo difusor, servidor o similares. Además, cada uno entre el dispositivo 12 de origen y el dispositivo receptor 14 puede incluir los componentes adecuados de modulación, demodulación, conversión de frecuencia, filtrado y amplificación, para la transmisión y recepción de vídeo codificado, según sea aplicable, incluso componentes inalámbricos de frecuencia de radio (RF) y antenas suficientes para dar soporte a la comunicación inalámbrica. Para facilitar la ilustración, sin embargo, tales componentes no se muestran en la FIG. 1.

Una secuencia de vídeo incluye una serie de tramas de vídeo. El codificador 20 de vídeo opera sobre bloques de píxeles dentro de tramas de vídeo individuales a fin de codificar los datos de vídeo. Los bloques de vídeo pueden tener tamaños fijos o variables, y pueden diferir en tamaño según un estándar de codificación especificado. Cada trama de vídeo incluye una serie de tajadas. Cada tajada puede incluir una serie de macrobloques, que pueden disponerse en subbloques. Como ejemplo, el estándar ITU-T H.264 da soporte a la intrapredicción en diversos tamaños de bloque, tales como 16 por 16, 8 por 8 y 4 por 4 para componentes luma, y 8x8 para componentes chroma, así como la interpredicción en diversos tamaños de bloque, tales como 16 por 16, 16 por 8, 8 por 16, 8 por 8, 8 por 4, 4 por 8 y 4 por 4 para componentes luma y los correspondientes tamaños ajustados para los componentes chroma.

Los bloques de vídeo más pequeños pueden proporcionar una mejor resolución y pueden usarse para ubicaciones de una trama de vídeo que incluyen mayores niveles de detalle. En general, los macrobloques (MB) y los diversos subbloques pueden considerarse como bloques de vídeo. Además, una tajada puede considerarse una serie de bloques de vídeo, tales como MB y / o subbloques. Cada tajada puede ser una unidad independientemente descodificable. Después de la predicción, puede efectuarse una transformación sobre el bloque residual de 8 x 8 o el bloque residual de 4 x 4, y puede aplicarse una transformación adicional a los coeficientes de DC (Comunicación de Datos) de los bloques de 4 x 4 para los componentes chroma o el componente luma si se usa la modalidad de intrapredicción de 16 x 16.

El codificador 20 de vídeo y / o el descodificador 26 de vídeo del sistema 10 de la FIG. 1 pueden configurarse para emplear técnicas para la codificación de longitud variable (VLC) adaptable, eficiente en términos de memoria y de baja complejidad, según lo descrito en la presente divulgación. En particular, el codificador 20 de vídeo y / o el descodificador 26 de vídeo pueden incluir, respectivamente, un codificador de entropía y un descodificador de entropía, que aplican al menos algunas de tales técnicas para reducir la utilización de memoria, el sobregasto de procesamiento, la complejidad de procesamiento, el consumo de ancho de banda, el espacio de almacenamiento de datos y / o el consumo de energía.

La FIG. 2 es un diagrama en bloques que ilustra un ejemplo de un codificador 20 de vídeo según se muestra en la FIG. 1. El codificador 20 de vídeo puede formarse, al menos en parte, como uno o más dispositivos de circuitos integrados, que pueden denominarse colectivamente un dispositivo de circuitos integrados. En algunos aspectos, el codificador 20 de vídeo puede formar parte de un dispositivo de mano de comunicación inalámbrica o servidor de difusión. El codificador 20 de vídeo puede efectuar la intracodificación y la intercodificación de bloques dentro de tramas de vídeo. La intracodificación se apoya en la predicción espacial para reducir o eliminar la redundancia espacial en el vídeo dentro de una trama de vídeo dada. La intercodificación se apoya en la predicción temporal para reducir o eliminar la redundancia temporal en el vídeo dentro de tramas adyacentes de una secuencia de vídeo. Para la intercodificación, el codificador 20 de vídeo realiza la estimación del movimiento para rastrear el movimiento de bloques de vídeo coincidentes entre tramas adyacentes.

Como se muestra en la FIG. 2, el codificador 20 de vídeo recibe un bloque 30 de vídeo actual dentro de una trama de vídeo a codificar. En el ejemplo de la FIG. 2, el codificador 20 de vídeo incluye la unidad 32 de estimación de movimiento, el almacén 34 de tramas de referencia, la unidad 36 de compensación de movimiento, la unidad 38 de transformación de bloques, la unidad 40 de cuantización, la unidad 42 de cuantización inversa, la unidad 44 de transformación inversa y la unidad 46 de codificación de entropía. La unidad 46 de codificación de entropía puede acceder a una o más estructuras de datos almacenadas en una memoria 45 para obtener datos útiles en la codificación. Un filtro de desbloqueo en el bucle (no mostrado) puede aplicarse para filtrar bloques, a fin de eliminar defectos visuales bloqueantes. El codificador 20 de vídeo también incluye el sumador 48 y el sumador 50. La FIG. 2 ilustra los componentes de predicción temporal del codificador 20 de vídeo para la intercodificación de bloques de vídeo. Aunque no se muestran en la FIG. 2 para facilitar la ilustración, el codificador 20 de vídeo también puede incluir componentes de predicción espacial para la intracodificación de algunos bloques de vídeo.

La unidad 32 de estimación de movimiento compara el bloque 30 de vídeo con bloques en una o más tramas de vídeo adyacentes, a fin de generar uno o más vectores de movimiento. La trama, o tramas, adyacente(s), puede(n) extraerse del almacén 34 de tramas de referencia, que puede comprender cualquier tipo de memoria o dispositivo de almacenamiento de datos para almacenar bloques de vídeo reconstruidos a partir de bloques previamente codificados. La estimación del movimiento puede llevarse a cabo para bloques de tamaños variables, p. ej., 16 x 16, 16 x 8, 8 x 16, 8 x 8 o tamaños de bloque más pequeños. La unidad 32 de estimación de movimiento identifica uno o más bloques en tramas adyacentes que coincida(n) más estrechamente con el bloque 30 de vídeo actual, p. ej., en base a un modelo de distorsión de velocidad, y determina el desplazamiento entre los bloques en tramas adyacentes y el bloque de vídeo actual. Sobre esta base, la unidad 32 de estimación produce uno o más vectores de movimiento (MV) que indican la magnitud y trayectoria del desplazamiento entre el bloque 30 de vídeo actual y uno o más bloques coincidentes de las tramas de referencia usadas para codificar el bloque 30 de vídeo actual.

Los vectores de movimiento pueden tener precisión de medio píxel o de un cuarto de píxel, o una precisión aún más

fina, permitiendo al codificador 20 de vídeo rastrear el movimiento con mayor precisión que las ubicaciones de píxeles enteros y obtener un mejor bloque de predicción. Cuando se usan los vectores de movimiento con valores de fracciones de píxel, las operaciones de interpolación se llevan a cabo en la unidad 36 de compensación de movimiento. La unidad 32 de estimación de movimiento identifica las mejores particiones de bloques y el vector de movimiento, o vectores de movimiento, para un bloque de vídeo, usando ciertos criterios, tales como un modelo de distorsión de velocidad. Por ejemplo, puede haber más de un vector de movimiento en el caso de la predicción bidireccional. Usando las particiones de bloques y los vectores de movimiento resultantes, la unidad 36 de compensación de movimiento forma un bloque de vídeo de predicción.

El codificador 20 de vídeo forma un bloque de vídeo residual restando el bloque de vídeo de predicción producido por la unidad 36 de compensación de movimiento al bloque 30 de vídeo actual original, en el sumador 48. La unidad 38 de transformación de bloques aplica una transformación, tal como la transformación entera de 4 x 4 o de 8 x 8 usada en el estándar H.264 / AVC, al bloque residual, produciendo coeficientes de bloque de transformación residual. La unidad 40 de cuantización cuantiza los coeficientes de bloque de transformación residual para reducir adicionalmente la tasa de bits. La unidad 46 de codificación de entropía codifica los coeficientes cuantizados para reducir incluso más la tasa de bits.

La unidad 46 de codificación de entropía funciona como una unidad de codificación de longitud variable (VLC) para aplicar la codificación VLC a los coeficientes de bloques cuantizados. En particular, la unidad 46 de codificación de entropía puede configurarse para efectuar la codificación de VLC de los coeficientes de bloques de vídeo digital usando técnicas de VLC adaptables, eficientes en términos de memoria y de baja complejidad, según lo descrito en la presente divulgación. Por tanto, los diversos procesos de codificación descritos en la presente divulgación pueden implementarse dentro de la unidad 46 de codificación de entropía para realizar la codificación de los datos de vídeo. Alternativamente, una tal unidad 46 de codificación de entropía puede llevar a cabo los procesos descritos en la presente divulgación para codificar cualquiera entre una gran variedad de datos, que incluye, sin limitarse a, los datos de vídeo, imagen, voz y audio. En general, el descodificador 26 de vídeo realiza operaciones inversas, incluyendo la descodificación VLC, para descodificar y reconstruir el vídeo codificado, como se describirá, p. ej., con referencia a la FIG. 3

La unidad 42 de cuantización inversa y la unidad 44 de transformación inversa aplican, respectivamente, la cuantización inversa y la transformación inversa para reconstruir el bloque residual. El sumador 50 añade el bloque residual reconstruido al bloque de predicción de movimiento compensado, producido por la unidad 36 de compensación de movimiento, a fin de producir un bloque de vídeo reconstruido para su almacenamiento en el almacén 34 de tramas de referencia. El bloque de vídeo reconstruido es usado por la unidad 32 de estimación de movimiento y la unidad 36 de compensación de movimiento para codificar un bloque en una subsiguiente trama de vídeo.

La FIG. 3 es un diagrama en bloques que ilustra un ejemplo de un descodificador 26 de vídeo. El descodificador 26 de vídeo puede formarse, al menos en parte, como uno o más dispositivos de circuitos integrados, que puede(n) denominarse colectivamente un dispositivo de circuitos integrados. En algunos aspectos, el descodificador 26 de vídeo puede formar parte de un equipo de mano de comunicación inalámbrica. El descodificador 26 de vídeo puede efectuar la intradescodificación y la interdescodificación de los bloques dentro de las tramas de vídeo. Como se muestra en la FIG. 3, el descodificador 26 de vídeo recibe un flujo de bits de vídeo codificado que ha sido codificado por el codificador 20 de vídeo. En el ejemplo de la FIG. 3, el descodificador 26 de vídeo incluye la unidad 52 de descodificación de entropía, la unidad 54 de compensación de movimiento, la unidad 56 de cuantización inversa, la unidad 58 de transformación inversa y el almacén 62 de tramas de referencia. La unidad 52 de descodificación de entropía puede acceder a una o más estructuras de datos almacenadas en una memoria 51 para obtener datos útiles en la codificación. El descodificador 26 de vídeo también puede incluir un filtro de desbloqueo en el bucle (no mostrado) que filtra la salida del sumador 64. El descodificador 26 de vídeo también incluye al sumador 64. La FIG. 3 ilustra los componentes de predicción temporal del descodificador 26 de vídeo para la interdescodificación de bloques de vídeo. Aunque no se muestran en la FIG. 3, el descodificador 26 de vídeo también puede incluir componentes de predicción espacial para la intradescodificación de algunos bloques de vídeo.

La unidad 52 de descodificación de entropía recibe el flujo de bits de vídeo codificado y descodifica, del flujo de bits, coeficientes residuales cuantizados, la modalidad de codificación de macrobloques e información de movimiento, que puede incluir vectores de movimiento y particiones de bloques. Por tanto, la unidad 52 de descodificación de entropía funciona como una unidad de descodificación de VLC. Por ejemplo, a fin de descodificar coeficientes residuales cuantizados del flujo de bits codificados, como la unidad 46 de codificación de entropía de la FIG. 2, la unidad 52 de descodificación de entropía de la FIG. 3 puede efectuar la descodificación VLC adaptable, eficiente en términos de memoria y de baja complejidad, de los coeficientes de bloques de vídeo digital, según lo descrito en la presente divulgación. Sin embargo, la unidad 52 de descodificación de entropía puede realizar la descodificación VLC de manera inversa con respecto a la unidad 46 de codificación de entropía de la FIG. 2, a fin de extraer los coeficientes de bloques cuantizados del flujo de bits codificados. Por tanto, los diversos procesos de

descodificación descritos en la presente divulgación pueden implementarse dentro de la unidad 52 de descodificación de entropía para realizar la descodificación de los datos de vídeo. Alternativamente, una tal unidad 52 de descodificación de entropía puede realizar los procesos descritos en la presente divulgación para descodificar cualquiera entre una gran variedad de datos, que incluye, sin limitarse a, datos de vídeo, imagen, voz y audio. En cualquier caso, el resultado de la codificación de longitud variable realizada por la unidad 52 de descodificación de entropía puede emitirse a un usuario, almacenarse en memoria y / o transmitirse a otro dispositivo o unidad de procesamiento.

La unidad 54 de compensación de movimiento recibe los vectores de movimiento y las particiones de bloques y una o más tramas de referencia reconstruidas desde el almacén 62 de tramas de referencia, para producir un bloque de vídeo de predicción. La unidad 56 de cuantización inversa cuantiza inversamente, es decir, decuantiza, los coeficientes de bloques cuantizados. La unidad 58 de transformación inversa aplica una transformación inversa, p. ej., una DCT (transformada de cosenos discretos) inversa o una transformación entera inversa de 4 x 4 o de 8 x 8, a los coeficientes, a fin de producir bloques residuales. Los bloques de vídeo de predicción son sumados luego por el sumador 64 a los bloques residuales para formar bloques descodificados. Un filtro de desbloqueo (no mostrado) puede aplicarse para filtrar los bloques descodificados, a fin de eliminar defectos visuales bloqueantes. Los bloques filtrados se colocan luego en el almacén 62 de tramas de referencia, que proporciona tramas de referencia para la descodificación de las tramas de vídeo subsiguientes y también produce vídeo descodificado para alimentar al dispositivo 28 de visualización (FIG. 1).

Codificación eficiente en términos de memoria de códigos de longitud variable

Un ejemplo de una técnica eficiente en términos de memoria para la codificación de longitud variable, a fin de dar soporte a estructuras de datos compactas según la presente invención se describirá ahora en mayor detalle. La técnica no debe necesariamente apoyarse en ningún esquema específico de construcción de código, tal como los de Huffman, Shannon, Shannon-Fano, Gilbert-Moore, u otros códigos. La técnica supone, sin embargo, que se construye un código para un origen con probabilidades monótonamente crecientes de símbolos. Más específicamente, se supone que las palabras de código tienen longitudes monótonamente decrecientes (o al menos no crecientes), y que las palabras de código de la misma longitud tienen el mismo orden lexicográfico que los símbolos en el alfabeto de entrada que representan.

Esta técnica, según se aplica a la codificación de vídeo o a otras aplicaciones, usa las propiedades anteriores para describir unívocamente un tal código con una estructura de datos muy compacta. Como se ha descrito anteriormente, la estructura de datos puede contener, para cada longitud válida de palabra de código, es decir, al nivel de nodos externos en un árbol de codificación, la siguiente información:

a. un valor parcial de la palabra de código lexicográficamente más pequeña (o más grande) en el nivel actual en un árbol de codificación,

b. un cierto número de bits en el valor parcial,

c. un valor de un símbolo correspondiente a la palabra de código lexicográficamente más pequeña (o más grande), y

d. un indicador que instruye a un descodificador para omitir un cierto número de bits antes de avanzar al siguiente nivel del árbol de codificación.

Los procesos de codificación y descodificación pueden usar esta estructura para identificar un nivel del árbol de códigos correspondiente a una palabra de código a producir (o descodificar), y luego calcular directamente el valor de la palabra de código (o símbolo descodificado).

Esta técnica puede permitir el uso de una cantidad mucho más pequeña de memoria para codificar y descodificar, debido al hecho de que sólo se almacenan valores parciales de las palabras de código lexicográficamente más pequeñas. La estructura de datos puede ser cualquiera entre una amplia variedad de estructuras de datos, tales como tablas, listas enlazadas, árboles binarios, árboles radicales, ficheros planos o similares, y puede almacenarse en cualquiera entre una gran variedad de distintos dispositivos de memoria, tales como las muchas formas de la memoria de acceso aleatorio (RAM), la memoria sólo de lectura (ROM), o ambas. La estructura de datos puede almacenarse en tal memoria dentro de un codificador o un descodificador, p. ej., dentro de la memoria 45 o la memoria 51 mostradas, respectivamente, en las FIGS. 2 y 3. Nuevamente, al menos algunos de los niveles del árbol de codificación incluyen palabras de código dispuestas en orden lexicográfico con respecto al orden de los valores de símbolos representados por las palabras de código. En consecuencia, cada una de las palabras de código básicas es una palabra de código lexicográficamente mínima en un nivel correspondiente en el árbol de codificación. Además, esta técnica permite el uso del acceso incremental a los datos del flujo de bits, permitiendo que un almacén temporal del flujo de bits esté representado por un registro razonablemente corto. Por ejemplo, un

registro de 32 bits podría ser suficiente, incluso para códigos muy largos. El registro puede actualizarse a intervalos convenientes (p. ej., de 8 bits), reduciendo adicionalmente la complejidad de la implementación. Globalmente, en diversos aspectos, la técnica puede ser capaz de reducir significativamente la complejidad de la representación y la codificación / descodificación de códigos de longitud variable, lo que puede permitir a los diseñadores de algoritmos de compresión utilizar libros de códigos mayores y más eficientes.

Se proporciona ahora una exposición general de los códigos de longitud variable, para asistir en la ilustración de las técnicas descritas en la presente divulgación. Los códigos de longitud variable representan una herramienta fundamental en la compresión de datos. En general, los códigos de longitud variable se usan para representar secuencias de símbolos con alguna distribución conocida y, habitualmente, sumamente desequilibrada. Tales secuencias pueden ser representadas por secuencias binarias, o códigos, de una longitud global mucho más corta. La reducción en la longitud se logra reemplazando los símbolos que ocurren más frecuentemente por códigos más cortos, y los símbolos menos frecuentes por códigos más largos.

Ejemplos de códigos de longitud variable usados en la compresión de datos con los *códigos de Huffman*, p. ej., según lo descrito en el documento de D. A. Huffman, “A method for the construction of minimum-redundancy codes” [“Un procedimiento para la construcción de códigos de redundancia mínima”], Proc. IRE, vol. 40, páginas 1098-1101, septiembre de 1952, los *códigos de Shannon*, p. ej., según lo descrito en el documento de C. E. Shannon, “A mathematical theory of communication” [“Una teoría matemática de la comunicación”], Bell Syst. Tech. J., Vol. 27, páginas 379-423, julio de 1948, los *códigos de Shannon-Fano*, p. ej., según lo descrito en el documento de R. M. Fano, “The transmission of information” [“La transmisión de información”], Res. Lab. Electronics, Instituto de Tecnología de Massachusetts, Cambridge, Mass., Inf. Téc. Nº 65, 1949, y los *códigos de Gilbert-Moore*, según lo descrito en el documento de E. N. Gilbert y E. F. Moore, “Variable-Length Binary Encodings” [“Codificaciones binarias de longitud variable”], Bell Syst. Tech. J., Vol. 7, páginas 932-967, 1959 (mencionados también a veces como los *códigos de Shannon-Fano-Elias*).

Los códigos descritos anteriormente pertenecen a una clase de *códigos libres de prefijos*, p. ej., según lo descrito en el documento de T. Cover y J. Thomas, “Elements of Information Theory” [“Elementos de la teoría de la información”], Wiley, 1991. La FIG. 4 es un diagrama que ilustra un ejemplo de un árbol de codificación binaria. Los códigos descritos anteriormente pueden representarse convenientemente por un árbol binario tal como el mostrado en la FIG. 4. Por tanto, un codificador puede codificar valores de símbolos coherentes con el árbol de codificación. Los valores según el árbol pueden representar a cualquiera entre una gran variedad de datos, tales como datos de vídeo, datos de imagen, datos de voz o datos de audio. Cada *nodo interno* en tal árbol corresponde a un dígito binario, cuyo valor 0 fuerza un movimiento hacia la derecha, y cuyo valor 1 fuerza un movimiento hacia el nodo descendiente izquierdo en un árbol. El nodo más alto se llama un nodo *raíz*, que es el nodo desde el cual comienza la codificación / descodificación.

Cada *nodo externo* en un árbol está donde se reinicia el proceso de codificación / descodificación, produciendo bien una palabra de código, es decir, como una secuencia de bits desde la raíz hasta el nodo actual, o bien un valor descodificado de un símbolo asociado a una palabra de código actual. En el ejemplo de árbol de codificación de la FIG. 4, hay dieciséis palabras de código correspondientes a símbolos indizados entre 0 y 15. En este ejemplo, la palabra de código más corta tiene una longitud de 1 bit, y las palabras de código más largas tienen longitudes de 10 bits cada una. El número de niveles que contienen nodos (palabras de código) externos en este árbol es 7, es decir, en los niveles 1º, 3º, 4º, 6º, 7º, 9º y 10º.

Con referencia adicional a la FIG. 4, sea n el número de nodos externos en el árbol de codificación (y, consecuentemente, el número de palabras de código en el código), sea L la longitud de la palabra de código más larga y sea K el número de niveles poblados con nodos externos en el árbol de codificación. La siguiente exposición usa la notación O de P. Bachmann. Por ejemplo, la expresión y(n) = O(x(n)) indica la existencia de alguna constante M > 0, tal que |y()| <= M |x(n)| para todo n suficientemente grande.

Para dar soporte al proceso de codificación, un codificador o descodificador, tal como la unidad 46 de codificación de entropía, o la unidad 52 de descodificación de entropía, generalmente necesita almacenar un árbol binario en la memoria del ordenador, tal como la memoria 45 o la memoria 51. Además, los procesos de codificación y descodificación deberían implicar un recorrido bit por bit (es decir, nodo por nodo) del árbol de codificación almacenado en la memoria. Por tanto, tal implementación debería implicar un coste O(n) de almacenamiento y puede llevar hasta O(L) etapas. No obstante, en algunos casos especiales, cuando los árboles de codificación tienen alguna estructura específica, puede haber maneras más efectivas de almacenar tales estructuras de código y de realizar las operaciones de codificación o descodificación.

Por ejemplo, considerando el código presentado en el ejemplo de árbol de codificación de la FIG. 4, puede observarse que las palabras de código son no decrecientes en longitud, y que todas las palabras de código en el mismo nivel del árbol de codificación tienen valores adyacentes. Por ejemplo, las palabras de código del 4º nivel del árbol de la FIG. 4 son más largas que las palabras de código en el 3er nivel del árbol, es decir, 0001 ante 011, 010,

001 y 000. Además, las palabras de código en el 3er nivel tienen valores adyacentes de 011, 010, 001 y 000. Por tanto, en lugar de almacenar todos los códigos, puede ser suficiente almacenar sólo la palabra de código más pequeña o más grande para cada nivel del árbol de codificación, es decir, como una palabra de código básica a partir de la cual puedan calcularse las palabras de código adyacentes.

La anterior observación es la clave a fin de entender las técnicas para la descodificación de códigos de longitud variable en base a su conversión a la denominada *forma canónica*, p. ej., según lo descrito en el documento de A. Moffat y A. Turpin, "On the implementation of Minimum-Redundancy Prefix Codes" ["Sobre la implementación de códigos de prefijo de redundancia mínima"], IEEE Trans. Communications, 45 (10) (1997) 1200-1207. En términos sencillos, un código canónico tiene una distribución no decreciente de longitudes y mantiene el orden lexicográfico con respecto a los índices asignados a sus nodos. Es bastante sencillo mostrar que cualquier origen dado puede reordenarse de modo tal que el código resultante tenga las propiedades anteriores.

Por ejemplo, el código ilustrado en el árbol de codificación de la FIG. 4 representa un código reordenado para un origen con una distribución no monótona, según lo indicado en la TABLA 1 a continuación. Específicamente, la TABLA 1 es un ejemplo de un código canónico de longitud variable que ha sido reordenado.

TABLA 1

Ejemplo de código canónico de longitud variable.				
Símbolo	**Probabilidad**	**Índice del símbolo después del reordenamiento**	**Longitud de código**	**Código**
0	0,6561	15	1	1
1	0,0729	12	3	011
2	0,0729	13	3	010
3	0,0081	6	7	0000101
4	0,0729	14	3	001
5	0,0081	6	7	0000100
6	0,0081	10	6	000011
7	0,0009	1	10	0000000001
8	0,0729	11	4	0001
9	0,0081	7	7	0000011
10	0,0081	8	7	0000010
11	0,0009	2	9	000000011
12	0,0081	9	7	0000001
13	0,0009	3	9	000000010
14	0,0009	4	9	000000001
15	0,0001	0	10	00000000000

En la TABLA 1 anterior, el símbolo 0 tiene la mayor probabilidad, seguido por el 1 y el 2. Sin embargo, el símbolo 3 tiene una probabilidad inferior al 4, y el 4 y el 8 tienen la misma probabilidad que el 1 y el 2. Después del reordenamiento, todas las probabilidades de símbolo son monótonamente crecientes (no decrecientes), y dan como resultado el código canónico mostrado representado en el árbol de codificación de la FIG. 4. El llamado algoritmo de Moffat-Turpin, según lo descrito en el documento de A. Moffat y A. Turpin, "On the Implementation of Minimum-Redundancy Prefix Codes" ["Sobre la implementación de códigos de prefijo de redundancia mínima"], IEEE Trans. Communications, 45 (10) (1997) 1200-1207, proporciona una técnica para descodificar códigos canónicos. Las técnicas descritas en la presente divulgación pueden proporcionar mejoras sobre el algoritmo de Moffat-Turpin. Otros algoritmos, tales como los descritos en los documentos de J. B. Connell "A Huffman-Shannon-Fano Code" ["Un código de Huffman-Shannon-Fano"], Proc. IEEE, julio de 1973, 1046-1047, y de A. Brodnik y S. Carlsson

"Sublinear Decoding of Huffman Codes Almost in Place" ["Descodificación sublineal de códigos de Huffman casi en su sitio"], Taller DIMACS sobre Códigos y Árboles: Enfoques teóricos algorítmicos y de información, octubre de 1998, Universidad de Rutgers, Centro DIMACS, NJ, son similares al algoritmo de Moffat-Turpin, y también pueden mejorarse usando las técnicas reveladas de manera similar.

El algoritmo de Moffat-Turpin para la descodificación de códigos de longitud variable se describe más adelante. Suponiendo que un alfabeto A de entrada contiene n letras: A = {α_0, ..., α_{n-1}}, se aplica el reordenamiento *i*: A -> {0, ..., n-1}, de modo tal que las probabilidades resultantes satisfagan: $p_0 \leq p_1 \leq ... \leq p_{n-1}$. Luego, puede aplicarse un algoritmo de Huffman u otro algoritmo de construcción de redundancia mínima, que asigne longitudes de palabras de código l_i para cada índice $1 \leq i \leq L$, donde L es la longitud de la palabra de código más larga. Como resultado, se producen "números de población" como m_l, es decir, el número de palabras de código de longitud *l*.

Usando estos parámetros, se calculan los llamados valores "básicos" para cada nivel en el árbol, de la siguiente manera:

$$base[l] = \left\lceil \frac{\sum_{k=l+1}^{L} m_k \, 2^{L-k}}{2^{L-l}} \right\rceil = \left\lceil \sum_{k=l+1}^{L} m_k \, 2^{l-k} \right\rceil, \quad (1 \leq l \leq L)$$

Estos valores de palabras de código básicas representan las palabras de código canónicas lexicográficamente más pequeñas en cada nivel del árbol. Dado un valor de palabra de código básica *base[l]*, puede calcularse el valor de la j+1-ésima palabra de código entre m_l palabras de código de longitud *l*:

$$base[l] + j.$$

Para la operación del descodificador, es más conveniente almacenar una versión alineada a izquierda del valor de la palabra de código básica, según lo siguiente:

$$lj\_base[l] = base[l]2^{W-l},$$

donde W es la longitud del almacén temporal de bits, o registro, usado para guardar los bits más recientes cargados desde el flujo de bits. Se supone que W >=L.

Finalmente, además de los valores de palabras de código básicas, el descodificador de Moffat-Turpin también almacena índices de los respectivos símbolos en la formación reordenada. Estos índices se almacenan como valores de desplazamiento[*l*] para cada nivel poblado en el árbol. La estructura ejemplar completa mantenida por el algoritmo de Moffat-Turpin para el código representado por el árbol de la FIG. 4 se presenta en la TABLA 2 a continuación.

TABLA 2

Estructura del descodificador de Moffat-Turpin para el código en la FIG. 4			
i	**Lj_base [i] (W=16)**	**Nivel[i]**	**desplazamiento [i]**
0	**1**000000000000000	1	15
1	**001**0000000000000	3	12
2	**0001**000000000000	4	11
3	**000011**0000000000	6	10
4	**0000001**000000000	7	5
5	**000000001**0000000	9	2
6	**0000000000**000000	10	0

Un ejemplo de seudocódigo para la implementación del algoritmo de descodificación de Moffat-Turpin, usando la estructura de la TABLA 2, se presenta a continuación en la TABLA 3.

TABLA 3

Algoritmo descodificador de Moffat-Turpin		
Línea	**Instrucción**	**Observación**
1	V = almacén_temporal_del_flujo_de_bits;	obtener los últimos W bits del flujo de bits
2	for (i = 0; i<K; i++) {	
3	if (lj_base[i] <= V)	buscar nivel
4	break;	que contiene la palabra de código actual
5	}	
6	*l* = nivel[*i*];	obtener longitud
7	desplazar_flujo_de_bits(*l*);	desplazar flujo de bits en *l* bits
8	símbolo = desplazamiento[i] + ((V-base[*i*] >> (W-*l*));	descodificar símbolo

De la TABLA 3 anterior, puede verse que el proceso entero de descodificación implica hasta K comparaciones (de W bits) del almacén temporal del flujo de bits actual con valores de palabras de código básicas alineadas a izquierda, seguidas por el cálculo directo sencillo de un índice de un símbolo descodificado. También puede verse que el vector lj_base[] utilizado por tal estructura requiere O(K*W) bits de memoria, lo que podría ser un problema si las palabras de código son largas, ya que debe usarse un W tal que W >=*l*.

En el ejemplo de la TABLA 3, un descodificador recibe W bits desde el flujo de bits como V, y compara V con las palabras de código básicas (lj_base[i]) para niveles *i* sucesivos del árbol de codificación. Cuando se halla una palabra de código básica que es menor o igual que V, termina la búsqueda para el nivel de la palabra de código. Luego, el descodificador determina la longitud asociada al nivel *i*, desplaza el flujo de bits en *l* bits, y descodifica el símbolo. En particular, el símbolo descodificado está determinado por la suma del valor de desplazamiento para el nivel *i* y la diferencia entre la palabra de código V del flujo de bits y la palabra de código básica para el nivel *i*, desplazado hacia la derecha en W-l bits.

En una configuración general, cuando se sigue la descodificación de Moffat-Turpin, la búsqueda de asociación inversa i^{-1}: {0, ..., n-1} -> A. En este caso, la última operación se convierte en la más cara en términos de memoria, ya que requiere O(n) espacio. En muchos casos prácticos, sin embargo, tales como las situaciones que implican longitudes de ristras o salidas de transformaciones o predictores, los orígenes que deben codificarse ya están ordenados. En consecuencia, la memoria usada por el vector lj_base[] en la estructura de Moffat-Turpin se convierte en el factor principal en coste global de almacenamiento.

Las técnicas descritas en la presente divulgación proporcionan refinamientos que permiten la compresión adicional de las estructuras de datos usadas en el algoritmo de Moffat-Turpin, u otros algoritmos, y dan soporte a la descodificación incremental de códigos de longitud variable. Los refinamientos se expondrán ahora en mayor detalle. Con referencia a la TABLA 2, es evidente que los valores de lj_base[*l*] tienen grandes cantidades de bits iniciales de derecha a izquierda. En consecuencia, los valores parciales de las palabras de código básicas representan la eliminación de un número fijo de bits iniciales de las palabras de código básicas. En la mayoría de los casos, los bits iniciales que se eliminan serán ceros. Tales números de ceros son monótonamente crecientes según el árbol de codificación se extiende hacia capas más profundas. Por tanto, si los bits se descodifican secuencialmente a partir de la primerísima capa del árbol de codificación y avanzando hacia abajo, es posible omitir algunos de los bits cero iniciales sin afectar la corrección de la descodificación. Omitiendo al menos algunos de los ceros iniciales, las técnicas descritas en la presente divulgación permiten estructuras de datos muy comprimidas y la descodificación incremental de códigos de longitud variable.

Cuando se eliminan los bits iniciales, sin embargo, es posible que algunos códigos legítimos en los niveles

inferiores del árbol de codificación puedan extenderse hacia la gama de bits iniciales que se eliminan. En consecuencia, para evitar perder tales códigos, se proporciona una tabla de indicadores de omisión. El indicador de omisión instruye a un descodificador para omitir un cierto número de bits en un flujo de bits a descodificar antes de avanzar a un nivel seleccionado del árbol de codificación. En particular, el indicador de omisión puede instruir al descodificador para omitir un número fijo de bits, p. ej., 8 bits, en el flujo de bits de palabras de código, antes de avanzar al nivel seleccionado del árbol de codificación. De esta manera, el valor parcial de la palabra de código básica en el nivel seleccionado del árbol se basa en un desplazamiento de la palabra de código básica en el número fijo de bits. Sin el desplazamiento, la palabra de código básica en el nivel seleccionado del árbol se extendería, al menos parcialmente, hacia el número eliminado de bits iniciales.

La TABLA 4 a continuación ilustra un ejemplo de implementación de un proceso de codificación, en el cual se eliminan los ceros iniciales, de acuerdo a un aspecto de la presente divulgación, para comprimir adicionalmente la estructura de datos usada para representar y procesar las palabras de código. En el ejemplo de la TABLA 4, los ceros iniciales se eliminan en incrementos de 8, lo que es conveniente para la gestión de almacenes temporales de flujos de bits en ordenadores convencionales de 8 bits por octeto. Para gestionar la eliminación de ceros iniciales, se proporciona una tabla adicional de indicadores (eliminar_8[i]), según lo anteriormente descrito. Por tanto, la TABLA 4 es generalmente conforme a la TABLA 2, pero elimina los ceros iniciales de cada una de las palabras de código y añade la columna del indicador de omisión.

TABLA 4

Estructura modificada del descodificador de Moffat-Turpin				
i	**r_li_base[i] (W=8)**	**eliminar_8[i]**	**r_nivel[i]**	**desplazamiento[i]**
0	**1**0000000	0	1	15
1	**001**00000	0	3	12
2	**0001**0000	0	4	11
3	**00001**100	0	6	10
4	**0000001**0	1	7	5
5	**1**0000000	0	9-8=1	2
6	**00**000000	0	10-8=2	0

En el ejemplo de la TABLA 4, el valor r_lj_base(i) representa el valor de la palabra de código básica en cada posición de índice, el valor r_nivel[i] indica el nivel dentro del árbol de codificación para la posición del índice y la longitud de las palabras de código en ese nivel, el valor desplazamiento[i] indica el número de ceros iniciales de derecha a izquierda para el valor de la palabra de código básica y el valor omitir_8[i] indica si un descodificador debería omitir 8 bits para el próximo valor de palabra de código, designando el 1 una omisión y designando el 0 ninguna omisión. Esta operación de omisión refresca periódicamente el almacén temporal de bits, a intervalos seleccionados, para permitir que el descodificador identifique palabras de código que se perderían en otro caso cuando se eliminan los ceros iniciales. Por ejemplo, si se eliminan los ocho ceros iniciales más a la derecha de una palabra de código alineada a izquierda, la palabra de código que se extiende hacia los ocho bits más a la derecha se perdería parcial o completamente. En consecuencia, omitir los ocho bits más a la izquierda en respuesta a la indicación de omisión movería la palabra de código hacia la gama de bits que no se eliminan, preservando por ello la palabra de código para su uso en la descodificación.

Por tanto, el indicador de omisión señala cuándo el descodificador debería saltar hacia adelante en el incremento de omisión especificado para el siguiente nivel del código, p. ej., 8 en el ejemplo de la TABLA 4. Como ilustración, en la TABLA 2, los valores de palabras de código básicas en las posiciones de índice 5 y 6 (niveles 9 y 10 del árbol) son, respectivamente, 0000000010000000 y 0000000000000000. Cuando los ocho ceros iniciales de más a la derecha (alineados a izquierda) se eliminan de estos valores de palabras de código básicas, es necesario que el descodificador salte hacia adelante ocho bits, de modo que el valor efectivo de la palabra de código básica (00000000100000000) no se pierda al eliminar ocho ceros iniciales. En cambio, el valor efectivo de la palabra de código básica (0000000010000000) se convierte en un valor distinto de palabra de código básica (10000000) omitiendo los primeros ocho bits (00000000) y eliminando luego los ocho ceros iniciales más a la derecha.

Debido a la eliminación de los ceros iniciales, el ancho del vector lj_base[i] modificado es mucho más pequeño. En el código de la TABLA 4, como ejemplo, el ancho W del vector lj_base[i] modificado es W=8, en lugar de W=16 en el caso de la TABLA 2. Un ejemplo de una implementación de un algoritmo que usa una tal tabla de omisión extra

para refrescar periódicamente el almacén temporal de bits se muestra a continuación en la TABLA 5. Un algoritmo construido según se muestra en la TABLA 5 puede configurarse para dar soporte a palabras de código muy largas o tablas muy compactas de valores (lj_base) de palabras de códigos básicas.

TABLA 5

Algoritmo modificado del descodificador de Moffat-Turpin		
Línea	**Instrucción**	**Observación**
1	V = almacén_temporal_del_flujo_de_bits;	obtener los últimos W bits del flujo de bits
2	for (i = 0; i<K; i++) {	
3	if (lj_base[i] <= V)	buscar nivel
4	break;	que contiene la palabra de código actual
5	if (omitir_B[i])	¿omitimos los siguientes B bits?
6	V = desplazar_flujo_de_bits(B);	Desplazar flujo de bits en B bits
7	}	
8	*l* = nivel[*i*];	obtener longitud residual
9	desplazar_flujo_de_bits(*l*);	Desplazar flujo de bits en *l* bits
10	símbolo = desplazamiento[i] + ((V-base[i] >> (W-l));	descodificar símbolo

Como se muestra en la TABLA 5, el descodificador obtiene los últimos W bits del flujo de bits, representados por el valor V = almacén_temporal_del_flujo_de_bits. El descodificador busca luego en los niveles *i* del árbol de codificación un valor lj_base[i] de palabra de código básica que sea menor o igual que la palabra de código V del almacén temporal del flujo de bits. Si el nivel actual *i* del árbol corresponde a un nivel de omisión (omisión_B[i]), p. ej., según lo indicado en la TABLA 5, entonces el descodificador desplaza el flujo de bits hacia la derecha en B bits, p. ej., 8 bits en algunas implementaciones, de modo que la palabra de código en el próximo nivel buscado por el descodificador pueda retenerse, en lugar de perderse, al eliminar los B ceros iniciales más a la derecha.

Al determinar la longitud residual *l* = nivel[*i*] para las palabras de código en el nivel actual del árbol, p. ej., según lo indicado en la TABLA 5, el descodificador desplaza el flujo de bits en la longitud *l*. Luego, el descodificador calcula directamente el índice de símbolo en base a la suma del desplazamiento para el nivel actual *i* y la diferencia entre el contenido V del almacén temporal del flujo de bits y la palabra de código básica para el nivel actual *i*, desplazado a la derecha en *W-l* bits.

El descodificador descodifica una palabra de código del flujo de bits de palabras de código, usando la estructura de datos almacenada que especifica los valores parciales de las palabras de código básicas, el indicador de omisión, los valores representados por la palabra de código básica y las longitudes (es decir, el número de bits) de los valores parciales de las palabras de código básicas. En general, un procesador asociado a un descodificador, tal como la unidad 52 de descodificación de entropía, busca en los niveles del árbol de codificación un valor seleccionado entre los valores parciales de las palabras de código básicas que sea menor o igual que la palabra de código del flujo de bits de palabras de código. El procesador omite un cierto número de bits en el flujo de bits de palabras de código antes de avanzar a un nivel seleccionado del árbol de codificación, en respuesta al indicador de omisión, y calcula uno entre una pluralidad de valores correspondientes a la palabra de código, en base a una diferencia entre el valor seleccionado entre los valores parciales de las palabras de código básicas, que sea menor o igual que la palabra de código, y la palabra de código, y un índice del valor seleccionado entre los valores parciales de las palabras de código básicas que sea menor o igual que la palabra de código. El procesador genera el resultado de la descodificación para el almacenamiento en la memoria, la transmisión a un dispositivo o unidad de procesamiento distinta, o la presentación a un usuario. Por ejemplo, los resultados descodificados pueden

usarse para controlar el dispositivo 28 de visualización a fin de presentar vídeo o imágenes, y / o un dispositivo de salida de audio para presentar salida de audio o de voz.

En el ejemplo de la TABLA 5, el descodificador realiza actualizaciones incrementales del almacén temporal de flujos de bits mediante la operación de omisión, para retener palabras de código que, de otro modo, se perderían cuando se eliminan los ceros iniciales. Además, los valores de palabras de código básicas que el descodificador compara en cada nivel del código pueden ser mucho más cortos. La magnitud potencial de la reducción en la longitud del valor de palabras de código básicas se expondrá ahora. A fin de analizar la gama dinámica de tales cantidades en el algoritmo modificado descrito en la presente divulgación, se considera la diferencia entre 2 niveles adyacentes, según lo siguiente:

$$\begin{aligned} lj\_base[l] - lj\_base[l+i] &= \left\lceil \sum_{k=l+1}^{L} m_k \, 2^{l-k} \right\rceil 2^{W-l} - \left\lceil \sum_{k=l+i+1}^{L} m_k \, 2^{l+i-k} \right\rceil 2^{W-l+i} \\ &\le 2^{W-l} + \sum_{k=l+1}^{L} m_k \, 2^{W-k} - \sum_{k=l+i+1}^{L} m_k \, 2^{W-k} \\ &= 2^{W-l} + \sum_{k=l+1}^{l+i+1} m_k \, 2^{W-k} \end{aligned}$$

Si *i* es el índice del siguiente nivel no vacío, entonces:

$$lj\_base[l] - lj\_base[l+i] = 2^{W-l} + m_{l+i} \, 2^{W-(l+i)}.$$

Aquí, la cantidad principal de interés es: $m_{l+i}2^{-i}$, que influye sobre esta diferencia. En el caso más sencillo, cuando *i* = 1, es claro que esta diferencia depende sencillamente del número de nodos externos y, por tanto, *W* puede escogerse tal que:

$$W \ge \max_{l} \log_2 (m_l),$$

que, en la mayor parte de los casos prácticos, es una cantidad significativamente más pequeña que L. Esta diferencia debería ser especialmente grande para distribuciones sumamente desequilibradas.

Por ejemplo, si los símbolos de entrada son bloques de *m* bits con probabilidades de Bernoulli $p^m(1-p)^{m-k}$, entonces los niveles más poblados deberían contener aproximadamente $\binom{m}{pm} \approx 2^{H(p)m}$ palabras de código, lo que significa que aproximadamente *H(p)m* bits deberían usarse para diferenciar entre palabras de código, donde H(p) es la función de entropía, p. ej., según lo descrito en la obra de T. Cover y J. Thomas, “Elements of Information Theory” [“Elementos de la teoría de la información”], Wiley, 1991.

Por otra parte, la palabra de código más larga en este caso tendrá aproximadamente $L \approx -\log\left(\min_k \left\{ p^m (1-p)^{m-k} \right\}\right) = -\log(p_{\min})m = H_{-\infty}(p)m$ bits, donde es bien conocido que, para distribuciones asimétricas:

$$H_{-\infty}(p) > H(p)$$

donde $H_{-\infty}(p)$ es un caso especial de la entropía de Renyi, p. ej., según lo descrito en la obra de W. Szpankowski, “Average Case Analysis of Algorithms on Sequences” [“Análisis de casos promedio de algoritmos sobre secuencias”] (Nueva York, John Wiley & Sons, 2001). Esta diferencia puede ser arbitrariamente grande con p ->0 o p->1.

En base a la exposición anterior, se deduce que la técnica propuesta debería ser efectiva para manipular grandes estructuras asimétricas de códigos. Tradicionalmente, es difícil trabajar con tales estructuras usando las técnicas tradicionales / existentes y, en muchos casos, los ingenieros recurren a usar diversas simplificaciones que afectan a las prestaciones de compresión de los códigos, para hacerlos más prácticos.

Por ejemplo, los muy populares códigos de Golomb, p. ej., según lo descrito en la obra de S. Golomb, “Run-length coding” [“Codificación de longitud de ristras”], IEEE Trans. Inform. Theory, vol IT-12, páginas 399-401, julio de 1966 y la de R. Gallager y D. Van Voorhis “Optimal source codes for geometrically distributed integer alphabets” [“Códigos óptimos de origen para alfabetos enteros geométricamente distribuidos”], IEEE Trans. Inform. Theory, vol. IT-21, páginas 228-230, marzo de 1975, representan códigos de longitud variable con una estructura especialmente sencilla, pero son óptimos sólo para una clase de distribuciones geométricas y sólo para valores numerables de parámetros de tales distribuciones. Los ingenieros tienden a usarlos incluso para distribuciones significativamente divergentes, motivados principalmente por consideraciones de complejidad. Tales soluciones pueden devenir tanto sub-óptimas como muy difíciles de extender o de modificar, debido a las restricciones implícitas de prestaciones de tales códigos.

Otra solución asociada al diseño de los códigos de Lynch-Davisson, según lo descrito en la obra de T. J. Lynch “Sequence time coding for data compression” [“Codificación de tiempo de secuencia para la compresión de datos”], Proc. IEEE (Lett.), 54 (1966) 1490-1491 y en la de L. D. Davisson “Comments on Sequence time coding for data compression” [“Observaciones sobre la codificación de tiempo de secuencia para la compresión de datos”], Proc. IEEE (Lett.), 54 (1966) 2010-2011, es dividir los códigos en dos partes donde sólo una primera de ellas está sujeta a la codificación de longitud variable y la restante se transmite como una extensión, usando un número fijo de bits. Lamentablemente, hay una pérdida de eficiencia en tal división, a veces de hasta 1,5 a 2 bits por símbolos.

Una versión más elaborada de la técnica de división del libro de códigos ha sido desarrollada bajo el nombre de *agrupación alfabética*, p. ej., según lo descrito en la obra de Boris Ryabko, Jaakko Astola y Karen Egiazarian “Fast Codes for Large Alphabets” [“Códigos rápidos para grandes alfabetos”], Communications in Information and Systems, v. 3, n. 2, páginas 139 – 152, y la de Boris Ryabko y Jorma Rissanen “Fast Adaptive Arithmetic Code for Large Alphabet Sources with Asymmetrical Distributions” [“Código aritmético adaptable rápido para grandes fuentes alfabéticas con distribuciones asimétricas”], IEEE Communications Letters, v. 7, nº 1, 2003, páginas 33 a 35. Sin embargo, este enfoque también tiene como desventaja alguna pérdida en la eficiencia de compresión.

A diferencia de las técnicas anteriormente mencionadas, las técnicas descritas en la presente divulgación pueden configurarse para preservar totalmente la estructura y la optimalidad del código y, por lo tanto, pueden ser una herramienta útil para una amplia variedad de aplicaciones prácticas en la compresión y codificación de datos, tal como en la codificación y descodificación de datos de vídeo, imagen, audio o habla digitales.

Codificación de bloques adaptable binaria

Un ejemplo de una técnica de baja complejidad para la codificación adaptable de longitud variable de secuencias binarias producidas por fuentes sin memoria, que puede usarse conjuntamente con la invención, se describirá ahora en mayor detalle. Esta técnica revelada puede implementar códigos de bloques universales construidos para un conjunto de contextos identificados por los números de bits no nulos en bits previos en una secuencia. Las estructuras de datos pueden ser cualesquiera entre una amplia variedad de estructuras de datos, tales como tablas, listas enlazadas, árboles binarios, árboles radicales, ficheros planos o similares, y pueden almacenarse en cualquiera entre una gran variedad de distintos dispositivos de memoria, tales como las muchas formas de la memoria de acceso aleatorio (RAM), la memoria de sólo lectura (ROM), o ambas. La estructura de datos puede almacenarse en tal memoria dentro de un codificador o un descodificador. Esta técnica para la codificación y descodificación adaptable de baja complejidad puede apoyarse, al menos en parte, en una fórmula para la redundancia asintótica de tales códigos, que refina la estimación descrita en la obra de R. E. Krichevskiy y V. K. Trofimov “The Performance of Universal Encoding” [“Las prestaciones de la codificación universal”], IEEE Trans. Information Theory, 27 (1981), 199 – 207.

Los algoritmos de compresión de datos convierten secuencias de entrada de bits con alguna distribución desconocida en un flujo de bits descodificable. La compresión de datos se usa, por ejemplo, en el diseño de códecs de imagen o vídeo, la codificación escalable (basada en tajadas de bits) del espectro en códecs de audio y en otras aplicaciones. En la mayoría de tales casos, los bits a codificar se toman a partir de valores producidos por diversas herramientas de procesamiento de señales, tales como las transformaciones, los filtros de predicción y similares, lo que significa que ya están bien decorrelacionadas y que la hipótesis de la falta de memoria de tal origen está justificada.

Las implementaciones más usualmente utilizadas de tales algoritmos adaptables binarios están habitualmente basadas en códigos aritméticos, con algunas aproximaciones y atajos aplicados para reducir su complejidad. Dos ejemplos bien conocidos de tales algoritmos son el algoritmo codificador-Q descrito en la obra de W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, Jr. y R. B. Arps “An overview of the basic principles of the Q-Coder adaptive binary arithmetic coder” [“Un panorama de los principios básicos del codificador aritmético binario adaptable codificador-Q”], IBM J. Res. Dev., 32 (6) (1988) 717, que se usa en el estándar de codificación de imagen JBIG, o el algoritmo CABAC descrito en la obra de D. Marpe, H. Schwartz y T. Wiegand “Context-Based Adaptive Binary Arithmetic Coding in the H.264 / AVC video compression standard” [“Codificación aritmética binaria adaptable

basada en el contexto en el estándar de compresión de vídeo H.264 / AVC"], IEEE Trans. on CSVT, 13(7):620 636, julio de 2003, que se usa en los estándares ITU-T H.264 / MPEG AVC para la compresión de vídeo.

Una implementación alternativa usa una formación de códigos de Huffman diseñados para varias densidades estimadas e indizados por los números de bits no nulos en bloques (contextos) anteriores en una secuencia. En términos tanto de eficiencia como de implementación, tal técnica puede lograr prestaciones de compresión deseables usando incluso bloques de bits de tamaño modesto, p. ej., n=8 ... 16, y usando las correspondientes cantidades de memoria, p. ej., 1,5 kbytes ... 5 kbytes.

Esta técnica puede considerarse en el contexto de un origen sin memoria que produce símbolos de un alfabeto binario {0, 1} con probabilidades *p* y *q = 1 – p*, según corresponda. Si *w* es una palabra de longitud *n* producida por este origen, entonces su probabilidad es:

$$\Pr(w) = p^k q^{n-k}, \qquad (1)$$

donde *k* indica el número de unos en esta palabra. El valor *k* también puede denominarse el *peso* de *w*.

Un *código de bloque* ϕ es una asociación inyectiva entre las palabras w de longitud |w| = n y las secuencias binarias (o *palabras de código*) ϕ (w):

$$\phi : \{0,1\}^n \dashrightarrow \{0,1\}^*,$$

donde las palabras de código ϕ(w) representan un conjunto unívocamente descodificable, p. ej., según lo descrito en la obra de T. M. Cover y J. M. Thomas "Elements of Information Theory" ["Elementos de la teoría de la información"] (John Wiley & Sons, Nueva York, 1991).

Habitualmente, cuando el origen (es decir, su probabilidad *p*) se conoce, tal código está diseñado para minimizar su longitud promedio o, en términos relativos, su redundancia promedio:

$$R_\phi(n,p) = \frac{1}{n} \sum_{|w|=n} \Pr(w)\,|\phi(w)| - H(p)\,.$$

En la ecuación anterior, *H(p) = -p log p – q log q* indica la *entropía* del origen.

Los ejemplos clásicos de códigos y algoritmos sugeridos para resolver este problema incluyen los códigos de Huffman, Shannon, Shannon-Fano y Gilbert-Moore, y sus variantes. Las prestaciones de tales códigos están bien estudiadas y también se ha informado de muchas técnicas útiles de implementación práctica para tales códigos.

Cuando el origen no se conoce, la mejor opción disponible es diseñar un código universal ϕ^* que minimice la redundancia del peor caso para una clase de orígenes, p. ej., según lo descrito en la obra de B. M. Fitingof "Optimal Coding in the Case of Unknown and Changing Message Statistics" ["Codificación óptima en el caso de estadísticas de mensajes desconocidas y cambiantes"], Probl. Inform. Transm., 2, (2) (1965) 3 {11 (en Ruso) 1-7 (Traduc. al inglés), la de L. D. Davisson "Universal Noiseless Coding" ["Codificación universal sin ruido"], IEEE Trans. Inform. Theory, 19 (6) (1973), 783-795 y la de R. E. Krichevskiy y V. K. Trofimov "The Performance of Universal Encoding" ["Las prestaciones de la codificación universal"], IEEE Trans. Information Theory, 27 (1981) 199-207, y según se indica a continuación:

$$R_{\phi^*}(n) = \inf_{\phi} \sup_{p} R_\phi(n,p)\,.$$

Un ejemplo de tal código puede construirse usando las siguientes estimaciones de las probabilidades de palabras:

$$P_{KT}(w) = \frac{\Gamma(k+1/2)\,\Gamma(n-k+1/2)}{\pi\,\Gamma(n+1)},$$

donde Γ(x) es una función Γ, *k* es el peso de la palabra *w* y *n* es su longitud. La fórmula anterior se describe en la obra de R. E. Krichevsky y V. K. Trofimov "The Performance of Universal Encoding" ["Las prestaciones de la codificación universal"], IEEE Trans. Information Theory, 27 (1981) 199-207, y garantiza convergencia uniforme (en p) a las verdaderas probabilidades según n se aproxima al infinito (n -> ∞).

En una situación en la cual no se conoce el valor exacto de un parámetro del origen, es posible acceder a una secuencia de símbolos *u* producidos por este origen en el pasado. Tal secuencia puede denominarse una muestra, y puede suponerse que tiene |*u*| = *t* bits de longitud. La tarea aquí es diseñar un conjunto de códigos ϕ^*_u, indizados por distintos valores de esta muestra, de forma tal que su redundancia promedio resultante en el peor caso sea mínima, según se indica a continuación:

$$R_{\phi^*_u}(n,t) = \inf_{\{\phi_u\}} \sup_{p} \sum_{|u|=t} \Pr(u) R_{\phi_u}(n,p).$$

Tales códigos se llaman códigos de bloque universales basados en muestras, o adaptables. En la presente divulgación, las implementaciones específicas de códigos de bloques adaptables se describen utilizando las siguientes estimaciones de probabilidades de palabras w, dada una muestra *u*:

$$P_{KT}(w|u) = \frac{P_{KT}(u\,w)}{P_{KT}(u)} = \frac{\Gamma(k+s+1/2)\,\Gamma(n+t-k-s+1/2)}{\Gamma(s+1/2)\,\Gamma(t-s+1/2)} \frac{\Gamma(t+1)}{\Gamma(n+1)},$$

(1)

donde *s* es el peso de una muestra *u* y *t* es su longitud.

El concepto y análisis de códigos basados en muestras, utilizando la función estimadora (1) inmediatamente anterior están descritos por R. E. Krichevskiy en la obra de R. E. Krichevskiy “Universal Data Compression and Retrieval” [“Compresión y recuperación universal de datos”] (Kluwer, Norwell, MA, 1993). La tasa media de redundancia de un código de bloque adaptable es, asintóticamente:

$$R_{\phi^*_u}(n,t) \sim \frac{1}{2n} \log \frac{n+t}{t}, \qquad (2)$$

donde *n* es un tamaño de bloque y *t* es el tamaño de las muestras.

A partir de la ecuación (2) anterior, es evidente que, usando muestras de longitud *t = O(n)*, es posible reducir la tasa de redundancia de tales códigos hasta *O(1/n)*, lo que coincide con el orden de la tasa de redundancia de los códigos de bloques para orígenes conocidos. Sin embargo, a fin de poder entender el potencial total de tales códigos, se necesita conocer una expresión más exacta de su redundancia, incluyendo términos afectados por la elección del algoritmo efectivo de construcción de código, tal como el de Huffman, Shannon, o similares.

La presente divulgación ofrece la siguiente refinación del teorema 2 de Krichevskiy. En particular, el teorema 1 más adelante refina el teorema de tasa media de redundancia para un código ϕ^*_u de bloques adaptable, según lo siguiente.

Teorema 1: La tasa media de redundancia de un código ϕ^*_u de bloques adaptable tiene el siguiente comportamiento asintótico (n, t -> ∞):

$$\begin{aligned} R_{\phi^*_u}(n,t,p) &= \sum_{|u|=t} \Pr(u) R_{\phi^*_u}(n,p) \\ &= \frac{1}{n}\left\{\frac{1}{2}\log\frac{t+n}{t} + \Delta_{\phi^*_u}(n,t,p) + \frac{1-4pq}{24pq}\frac{n}{t(t+n)} - \frac{1-3pq}{24p^2q^2}\frac{(n+2t)\,n}{t^2(t+n)^2} \right. \\ &\qquad \left. + O\left(\frac{1}{t^3}+\frac{1}{n^3}\right)\right\}, \end{aligned}$$

(3A)

donde *n* es un tamaño de bloque y *t* es un tamaño de muestra, y *p, q = 1-p* son probabilidades de símbolos del origen de entrada, y donde:

$$\Delta_{\phi_u^*}(n,t,p) = \sum_{|u|=t} \sum_{|w|=n} \Pr(u) \Pr(w) \left[|\phi_u(w)| + \log P_{KT}(w|u) \right]$$

es la redundancia media del código ϕ^*_u con respecto a la distribución estimada en la ecuación (1) anterior.

El comportamiento exacto de $\Delta_{\phi_u^*}(n,t,p)$ es específico del algoritmo. Sin embargo, para una gran clase de técnicas de redundancia mínima, que incluye los códigos convencionales de Huffman y Shannon, este término está acotado en magnitud según lo siguiente:

$$|\Delta(n,t,S)| \leqslant 1,$$

y exhibe un comportamiento oscilante, que puede o no ser convergente con alguna constante, según el valor del parámetro *p*. Además, para valores cortos de *t* y *n*, la redundancia de tales códigos puede ser afectada por el próximo término siguiente:

$$\frac{1-4pq}{24pq} \frac{n}{t(t+n)}$$

que es una función del parámetro del origen *p*. La FIG. 5 es un gráfico que ilustra la tasa de redundancia de un código de bloque adaptable con un comportamiento asintótico, y grafica esta cantidad. Para bloques / muestras breves, las prestaciones de tales códigos se hacen sensibles a la asimetría del origen. La prueba de este teorema puede hallarse, por ejemplo, en la obra “Asymptotic average redundancy of adaptive block codes” [“Redundancia media asintótica de códigos de bloques adaptables”], de Y. A. Reznik y W. Szpankowski, Proceedings of IEEE International Symposium on Information Theory (ISIT) [Anales del Simposio internacional del IEEE sobre teoría de la información], 2003.

Se describirán ahora ejemplos de algoritmos eficientes para implementar los códigos descritos anteriormente. En un modelo sin memoria, la probabilidad de una palabra w (o de su estimación) depende sólo de su peso *k*, pero no de un patrón efectivo de sus bits. Por tanto, considerando un conjunto de todas las posibles palabras de n bits, podemos dividir el conjunto en n + 1 grupos:

$$\{0,1\}^n = W_{n,0} \cup W_{n,1} \cup \ldots \cup W_{n,k} \cup \ldots \cup W_{n,n},$$

que contienen palabras del mismo peso (*k*=0, ..., *n*) y la misma probabilidad. Los tamaños de tales grupos son $|W_{n,k}| = \binom{n}{k}$. Para mayor comodidad, se supone que cada grupo $W_{n,k}$ almacena las palabras en orden lexicográfico. El valor $t_{n,k}(w)$ indica el índice (posición) de una palabra w en un grupo $W_{n,k}$. La TABLA 6 a continuación es un ejemplo de un código construido para bloques de 4 bits con probabilidades de Bernoulli: $p^k q^{n-k}$, *p*=0,9.

TABLA 6

Ejemplo de código construido para bloques de 4 bits con probabilidades de Bernoulli: $p^k q^{n-k}$, p=0,9						
Bloque *w*	*k*	$l_{n,k}(w)$	Pr(w)	**Longitud**	Código ϕ(w)	**Subgrupo**
0000	0	0	0,6561	1	1	0
0001	1	0	0,0729	3	001	1
0010	1	1	0,0729	3	010	1
0011	2	0	0,0081	6	000011	3
0100	1	2	0,0729	3	011	1

0101	2	1	0,0081	7	0000001	4
0110	2	2	0,0081	7	0000010	4
0111	3	0	0,0009	9	000000001	5
1000	1	3	0,0729	4	0001	2
1001	2	3	0,0081	7	00000011	4
1010	2	4	0,0081	7	0000100	4
1011	3	1	0,0009	9	000000010	5
1100	2	5	0,0081	7	0000101	4
1101	3	2	0,0009	9	000000011	5
1110	3	3	0,0009	10	0000000001	6
1111	4	0	0,0001	10	0000000000	7

El ejemplo de código en la TABLA 6 se usará para describir la estructura de una asociación propuesta entre las palabras en el grupo $W_{n,k}$ y sus palabras de código, según ciertos aspectos de la presente divulgación. El código fue construido usando una modificación del algoritmo de Huffman, en el cual se emprendieron pasos adicionales para garantizar que las palabras de código situadas en el mismo nivel tengan el mismo orden lexicográfico que los bloques de entrada que representan. Es bien conocido que tal reordenamiento es posible sin ninguna pérdida de eficiencia de compresión. Los ejemplos de algoritmos anteriores que han usado este concepto de reordenamiento incluyen los códigos de Huffman-Shannon-Fano y los códigos canónicos descritos por Moffat y Turpin.

La FIG. 6 es un diagrama que ilustra un árbol de codificación que muestra la estructura del ejemplo de código de bloque de la TABLA 6. Como se espera, cada grupo $W_{n,k}$ consiste en, a lo sumo, dos subgrupos que contienen palabras de código de la misma longitud. En general, la estructura de código representada por el árbol de codificación de la FIG. 6 y los códigos de bloque de la TABLA 6 definen grupos de palabras de código, y subgrupos primeros y segundos de palabras de código dentro de cada uno de los grupos. Cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos. El primer subgrupo incluye palabras de código con una primera longitud y el segundo subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud. Las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código, para facilitar la codificación y descodificación por cálculo directo.

Un ejemplo de un grupo está indicado por el número 66 de referencia en la FIG. 6. Los ejemplos de subgrupos primeros y segundos están indicados por los números 68A, 68B de referencia, respectivamente, en la FIG. 6. Se proporcionan grupos y subgrupos similares para cada peso dentro del árbol de codificación. Un grupo contiene bloques con el mismo peso k. Un subgrupo contiene bloques con el mismo peso y el mismo nivel en el árbol de codificación. Esto se deduce del hecho de que todas las palabras en un grupo $W_{n,k}$ tienen la misma probabilidad y la denominada propiedad fraternal de los códigos de Huffman. Esta observación también vale para los códigos de Shannon, los códigos generalizados de Shannon y, posiblemente, otros algoritmos. Como se ha mencionado anteriormente, un grupo $W_{n,k}$ incluye a lo sumo dos subgrupos que contienen palabras de código de la misma longitud, y pueden representarse como:

$$W_{n,k} = W_{n,k,\ell} \cup W_{n,k,\ell+1} ,$$

donde ℓ es la longitud de código más corta que puede asignarse a los bloques del grupo $W_{n,k}$. Además, debido a que las palabras dentro del grupo $W_{n,k}$ siguen el orden lexicográfico, la división entre $W_{n,k,l}$ y $W_{n,k,l+1}$ es simplemente:

$$W_{n,k,\ell} = \{w \in W_{n,k} : I_{n,k}(w) < n_k\} ,$$

$$W_{n,k,\ell+1} = \{w \in W_{n,k} : I_{n,k}(w) \geq n_k\} ,$$

donde n_k indica el tamaño de un subgrupo con palabras de código más cortas. Por tanto, si un primer subgrupo tiene tres palabras de código con una longitud de 3 y un segundo subgrupo en el mismo grupo tiene una palabra de código con una longitud de 4, entonces n_k (el tamaño del subgrupo con las palabras de código más cortas, es decir, el primer subgrupo) es igual a 3. Este ejemplo corresponde a los subgrupos en el grupo asociado a los niveles 3 y 4 del árbol de codificación de la FIG. 6, donde el subgrupo 68A tiene palabras de código 001, 010 y 011 con longitudes de tres cada una, y el subgrupo 68B tiene la palabra de código 0001 con una longitud de cuatro.

Las palabras de código lexicográficamente más pequeñas en cada subgrupo pueden denominarse palabras de código básicas, p. ej., según lo descrito anteriormente con respecto a un primer aspecto de la presente divulgación, y pueden representarse como:

$$B_{n,k,l} = \phi(w_0),$$

$$B_{n,k,l+1} = \phi(w_{n_k}),$$

donde w_i es el i-ésimo bloque en el grupo $W_{n,k}$. Obsérvese que, según lo explicado anteriormente, las palabras de código restantes en ambos subgrupos pueden calcularse de la siguiente manera:

$$\phi(w_i) = \begin{cases} B_{n,k,l} + i, & \text{if } i < n_k, \\ B_{n,k,l+1} + i - n_k, & \text{if } i \geq n_k. \end{cases}$$

Como ilustración, se supone que hay un primer subgrupo 68A con tres palabras de código de longitud 3 y un segundo subgrupo 68B con una palabra de código de longitud 4, p. ej., como en el ejemplo de los niveles 3 y 4 del árbol de codificación de la FIG. 6. En este caso, si la posición de un bloque dado es $i = 2$, entonces $i < n_k$ (siendo n_k igual a 3), y la palabra de código resultante es la palabra de código básica aplicable + i. En este ejemplo, la palabra de código básica para el subgrupo es 001, y la palabra de código resultante es 001 +2 = 011. Para los niveles 3 y 4 del árbol de codificación de la FIG. 6, si la posición de la palabra de código aplicable era $i >= n_k$, entonces la palabra de código estaría en el segundo subgrupo y sería igual a la palabra de código básica de 0000 + i-n_k, que es igual a 0000 + 4 – 3 = 0001.

Las palabras de código básicas están sólo definidas por subgrupos no vacíos y el número de tales subgrupos S en un árbol construido para bloques de n bits está entre:

$$n + 1 \leq S \leq 2n.$$

Además, múltiples subgrupos pueden residir en el mismo nivel y el número de tales subgrupos cosituados no puede ser mayor que $n+1$. En el 10º nivel del árbol en la FIG. 6, por ejemplo, hay dos subgrupos correspondientes a las palabras de código 1110 y 1111. Sin embargo, estos subgrupos no pertenecen al mismo grupo. Esta es una diferencia significativa con respecto a otros algoritmos, que asignan palabras de código básicas únicas para cada nivel, pero requieren luego una tabla de reordenamiento de tamaño $O(n2^n)$ para trabajar con tales códigos. En el caso actual, la estructura entera tiene $O(n^2)$ bits.

En general, esta estructura de código define grupos W y subgrupos S. Cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos. Las palabras de código en cada uno de los grupos están ordenadas lexicográficamente con respecto a los valores representados por las palabras de código. Además, el primer subgrupo en cada grupo incluye palabras de código con una primera longitud y el segundo grupo incluye palabras de código con una segunda longitud, distinta a la primera longitud. La estructura de códigos puede ser representada por una estructura de datos a la que puede acceder un codificador para realizar la codificación de longitud variable. La presente divulgación contempla el uso de una tal estructura de códigos en la codificación o descodificación de longitud variable, así como un medio legible por ordenador que comprende una estructura de datos que define una tal estructura de códigos.

Dada la exposición anterior, se describirá ahora un algoritmo sencillo para el cálculo directo de códigos de bloques. Se supone que se dispone de los parámetros n_k ($0 \leq k \leq n$), y que puede obtenerse un nivel l y una palabra de código básica $B_{n.k.l}$ para cada subgrupo no vacío. Luego, el proceso de codificar un bloque w puede ser llevado a cabo, esencialmente, por un conjunto de las siguientes etapas:

1. Dado un bloque w, obtener su peso k e índice $I_{n.k}(w)$.

2. Si $I_{n.k}(w)$ < nk, seleccionar el primer subgrupo $W_{n.k.l}$; en caso contrario, seleccionar el segundo subgrupo $W_{n.k.l+1}$.

3. Luego, extraer la palabra de código básica ($B_{n.k.l}$ o $B_{n.k.l-1}$) para el subgrupo seleccionado ($W_{n.k.l}$ o $W_{n.k.l+1}$) y calcular el código pertinente según la siguiente ecuación:

$$\phi(w_i) = \begin{cases} B_{n,k,\ell} + i, & \text{if } i < n_k, \\ B_{n,k,\ell+1} + i - n_k, & \text{if } i \geq n_k. \end{cases} \qquad (13)$$

Según la ecuación anterior, si la posición $i = I_{n,k}(w)$ del bloque w en el subgrupo seleccionado ($W_{n.k.l}$ o $W_{n.k.l+1}$) es menor que el número n_k de bloques en el subgrupo, entonces la palabra de código es $B_{n.k.i} + i$. Alternativamente, si la posición i del bloque w en el subgrupo seleccionado ($W_{n.k.l}$ o $W_{n.k.l+1}$) es mayor o igual que el número n_k de bloques en el subgrupo, entonces la palabra de código es $B_{n.k.i+1} + i - n_k$.

Como se ha descrito anteriormente, como ilustración, para los niveles 3 y 4 del árbol de codificación de la FIG. 6, el proceso anterior produce una palabra de código de 011 cuando la posición de un código de bloque dado es $i = 2 < n_k$, y una palabra de código de 0001 cuando la posición de un código de bloque dado es $i = 3 > = n_k$. En este ejemplo, n_k es 3, es decir, el número de palabras de código en el primer subgrupo 68A para el peso k = 1. El orden de la posición i avanza lexicográficamente a partir de la palabra de código básica, p. ej., desde 0 a 3 en el caso del peso k = 1 en el ejemplo de la FIG. 6. En particular, la posición 0 corresponde a la palabra de código básica 001, la posición 1 corresponde a la palabra de código 010, y la posición 2 corresponde a la palabra de código 011, todas ellas en el primer subgrupo 68A ($i < n_k$) y la posición 3 corresponde a la palabra de código 0001 en el subgrupo 68B ($i >= n_k$).

Este proceso puede llevarse a cabo inmediatamente garantizando que las palabras de código situadas en el mismo nivel estén reordenadas de modo que tengan el mismo orden lexicográfico que los bloques de entrada que representan. Por ejemplo, las palabras de código descritas anteriormente siguen el orden lexicográfico de los bloques de entrada 0001, 0010, 0100 y 1000. Luego, las palabras de código lexicográficamente más pequeñas en cada subgrupo, p. ej., 001 en el subgrupo 68A o 0001 en el subgrupo 68B, pueden usarse como palabras de código básicas a los fines del cálculo de palabras de código descrito anteriormente. Un código de programa en lenguaje C, que representa un ejemplo de implementación de un proceso para la construcción directa de códigos de bloques, según lo descrito anteriormente, se expone en la TABLA 7 a continuación.

TABLA 7

Proceso para la construcción directa de códigos de bloques

```
/* encoder structure: */
typedef struct {
  unsigned short  nk[N+1];        /* # of elements in first (n,k) subgroup */
  unsigned char   sg[N+1][2];     /* (k,j) -> subgroup index mapping */
  unsigned char   len[S];         /* subgroup -> code length mapping */
  unsigned int    base[S];        /* subgroup -> base codeword mapping */
} ENC;

/* block encoder: */
unsigned block_enc (unsigned w, ENC *enc, BITSTREAM *bs)
{
  unsigned i, j, k, len, code;

  k = weight(w);                  /* split w into (k,index) */
  i = index(n,k,w);
  if (i >= enc->nk[k]) {          /* find subgroup containing w */
    i -= enc->nk[k];              /* adjust index */
    j = enc->sg[k][1];
  } else
    j = enc->sg[k][0];
  code = enc->base[j] + i;        /* generate code */
  len = enc->len[j];
  put_bits(code, len, bs);        /* write code to bitstream */

  return k;
}
```

En el código anterior en lenguaje C, el valor k indica el peso de un bloque w, el valor *i* indica la posición ($l_{n,k}(w)$) del bloque dentro de un grupo con peso *k* y *nk[k]* indica el número de palabras de código en el primer subgrupo del grupo con peso *k*. Si *i* es mayor o igual que *nk[k]*, entonces *i* se decrementa para ajustar el índice, y el subgrupo se fija en el segundo subgrupo (1) para el peso aplicable *k*. Este segundo subgrupo está identificado por *j = sg[k]*[1]. Si *i* es menor que *nk[k]*, *i* no se decrementa y el subgrupo se fija en el primer subgrupo (0) para el peso aplicable *k*. Este primer subgrupo está identificado por *j = sg[k]*[0].

Luego, la palabra de código se genera como una suma de la palabra de código básica para el subgrupo aplicable *j (base[j])* y el valor de *i*. Para el ejemplo de la FIG. 6, si el valor de *i* es 2, entonces el código sería la suma de la palabra de código básica 001 para el subgrupo 68A y el valor de *i* (2), que es igual a 001 + 010 = 011. Con referencia a la ecuación (13) anterior, según el subgrupo, la palabra de código básica es bien $B_{n,k,l}$ o $B_{n,k,l+1}$, y el valor de *i* es bien *i* o bien *i – nk[k]*. Por tanto, el código anterior generalmente corresponde al resultado proporcionado por la ecuación (13). Al calcular la palabra de código (*code*), la longitud (*len*) de la palabra de código se especifica como *len[j]*, que es la longitud del código para el subgrupo adecuado, donde el segundo subgrupo tiene una longitud de código que es mayor en una unidad a la del primer subgrupo. Luego, el proceso de codificación graba los códigos en el flujo de bits mediante la operación *put_bits (code, len, bs)*, que graba el valor de *code* y *len* en el flujo *bs* de bits. El flujo de bits se transmite para su descodificación por otro dispositivo. El proceso devuelve el peso *k* para el cálculo de la próxima palabra de código.

El proceso de codificación esbozado anteriormente puede implicar seleccionar uno de los grupos en base al peso del valor a codificar, seleccionar uno de los subgrupos en base a la posición lexicográfica del valor a codificar con respecto al número de palabras de código en el primer subgrupo del grupo seleccionado, seleccionar una de las palabras de código en el subgrupo seleccionado en base a la palabra de código básica para el subgrupo seleccionado y la posición lexicográfica del valor a codificar y codificar el valor a codificar con la palabra de código seleccionada. Las palabras de código básicas para cada uno de los subgrupos, las posiciones de las palabras de código dentro de cada uno de los grupos, el número de palabras de código dentro de cada uno de los primeros subgrupos y las longitudes de las palabras de código dentro de cada uno de los subgrupos pueden almacenarse en

una estructura de datos a la que un codificador puede acceder para dar soporte a la codificación de longitud variable.

Desde el punto de vista de la memoria, un proceso como el mostrado en la TABLA 7 sólo necesita *S* palabras de código básicas (de *O(n)* bits de largo), *n + 1* valores nk (de *O(n)* bits de largo), *S* longitudes de código (de *O(log n)* bits de largo) y *2(n + 1)* índices de subgrupo (de *O(log n)* bits de largo). La reducción adicional de la memoria es posible almacenando valores incrementales de palabras de código básicas, según lo expuesto en otra parte en la presente divulgación. Dado que *S = O(n)*, la entera estructura de datos necesita sólo $O(n^2)$ bits. En una implementación específica mostrada en la TABLA 7, suponiendo, p. ej., que *n – 20* y *S = 32*, el tamaño de esta estructura de datos resulta ser de 244 octetos. Esto es mucho menos que las 220 palabras que se necesitarían para presentar este código en forma de una tabla directa. Para bloques razonablemente cortos, p. ej., *n* <= 12 ... 16, el cálculo de pesos e índices (funciones weight(.) e index(...) en el proceso de la TABLA 7) puede ser cuestión de una simple consulta. En este caso, el proceso entero de codificación puede necesitar a lo sumo una comparación, dos sumas y cuatro consultas.

Para bloques mayores, puede usarse la siguiente fórmula combinatoria, bien conocida:

$$I_{n,k}(w) = \sum_{j=1}^{n} w_j \binom{n-j}{\sum_{k=j}^{n} w_k},$$

donde w_j representan bits individuales de la palabra *w*, y se supone que $\binom{n}{k} = 0$ para todo $k > n$. A fin de implementar esta fórmula, se podrían precalcular todos los coeficientes binomiales hasta el nivel *n* en el triángulo de Pascal, o bien calcularlos dinámicamente, usando las siguientes igualdades sencillas:

$$\binom{n-k}{k-1} = \frac{k}{n}\binom{n}{k}, \quad \text{y} \quad \binom{n-k}{k} = \frac{n-k}{n}\binom{n}{k}.$$

La implementación basada en coeficientes precalculados requiere $\frac{n(n+1)}{2} = O(n^2)$ palabras (*O(n3)* bits) de memoria y *O(n)* sumas. El cálculo dinámico de los coeficientes requerirá *O(n)* sumas, multiplicaciones y divisiones. Sin embargo, el proceso entero puede requerir sólo unos pocos registros y nada de memoria estática. Una exposición adicional de la complejidad del cálculo de índices puede hallarse en la obra de T. Tjalkens "Implementation cost of the Huffman-Shannon-Fano code" ["Coste de implementación del código de Huffman-Shannon-Fano"], en Proc. Data Compression Conference (DCC'05) (Snowbird, Utah, 29 al 31 de marzo, 2005) 123-132.

Se describirá ahora un ejemplo del diseño de un descodificador que implementa las técnicas anteriores. Igual que un proceso de codificación, según lo anteriormente descrito, un proceso de descodificación puede hacer uso de parámetros n_k, palabras de código básicas y longitudes asociadas. Para mayor comodidad, la siguiente exposición se apoyará en versiones alineadas a izquierda de los valores de palabras de código básicas:

$$B^{lj}_{n,k,\ell} = B_{n,k,\ell}\, 2^{T-\ell},$$

donde *T* es la longitud de una palabra de máquina (*T > max l*). Luego, puede describirse un proceso ejemplar de descodificación según lo siguiente:

1. Hallar el subgrupo más superior, siendo $B^{lj}_{n,k,\ell}$ menor que T bits en el flujo de bits;

2. Descodificar el índice de un bloque $I_{n,k}$ (w) en base a la ecuación (13) anterior; y

3. Producir un bloque reconstruido usando su peso k e índice.

El código en lenguaje C, que representa un ejemplo de implementación de un proceso de descodificación según lo anteriormente descrito, se proporciona en la TABLA 8 a continuación.

TABLA 8

Descodificación de códigos de bloque

```
/* decoder structure: */
typedef struct {
  unsigned short  nk[N+1];          /* # of elements in first (n,k) subgroup */
  struct {unsigned char k:7,j:1;} kj[S]; /* subgroup -> (k,j) mapping */
  unsigned char   len[S];           /* subgroup -> code length mapping */
  unsigned int    lj_base[S];       /* subgroup -> left-justified codewords */
} DEC;

/* block decoder: */
unsigned block_dec (unsigned *w, DEC *dec, BITSTREAM *bs)
{
  unsigned i, j, k, len, val;

  val = bitstream_buffer(bs);
  for (j=0; dec->lj_base[j]>val; j++) ; /* find a subgroup */
  len = dec->len[j];
  scroll_bitstream(len, bs);        /* skip decoded bits  */
  i = (val - dec->lj_base[j]) >> (32-len);
  k = dec->kj[j].k;                 /* get weight */
  j = dec->kj[j].j;                 /* get sub-group index */
  if (j)                            /* reconstruct index */
    i += dec->nk[k];
  *w = word(n,k,i);                 /* generate i-th word in (n,k) group */

  return k;
}
```

El ejemplo de proceso de descodificación mostrado en la TABLA 8 hace uso de palabras de código básicas *lj_base[S]* alineadas a izquierda. En funcionamiento, el proceso de descodificación recibe el contenido *val* del almacén temporal del flujo de bits e identifica un subgrupo dentro del árbol de codificación que tiene una palabra de código básica correspondiente al contenido *val* del almacén temporal del flujo de bits. Por ejemplo, el proceso continúa recorriendo hacia abajo a través de los subgrupos a distintos niveles en el árbol de codificación, mientras las palabras de código básicas sean mayores que la palabra de código *val* recibida. Cuando se llega a un subgrupo con una palabra de código básica que sea menor o igual a *val*, sin embargo, se selecciona ese subgrupo. Al hallar el subgrupo adecuado, el proceso determina la longitud del código para el subgrupo y luego desplaza el flujo de bits en esa longitud, para omitir los bits descodificados y aislar la palabra de código. El proceso de descodificación determina la posición del índice *i* de la palabra de código dentro del subgrupo, restando el valor de la palabra de código básica al contenido del almacén temporal del flujo de bits.

Si la palabra de código es 011 y la palabra de código básica es 010, por ejemplo, entonces el resultado de esta diferencia es 2, lo que indica que la palabra de código está en la posición 2 entre las posibles posiciones 0, 1 y 2 en el subgrupo. Para el ejemplo de un registro de 32 bits de ancho, esta diferencia puede desplazarse a la derecha en 32 unidades, menos la longitud *len* del código. El proceso de descodificación recupera luego el peso pertinente *k* y el índice *j* de subgrupo, y reconstruye el índice *i*. El proceso genera luego la *i*-ésima palabra en el grupo seleccionado como la palabra de código y devuelve el peso *k*. La expresión kj[j].k devuelve el peso del subgrupo, y la expresión kj[j].j devuelve el índice del subgrupo bien como un 0 o bien como un 1, lo que indica bien el primer subgrupo (0) o bien el segundo subgrupo (1) para el peso dado. Si se selecciona el segundo subgrupo de modo que *j* = 1, entonces el índice *i* se ajusta añadiendo el valor de nk[k]. En caso contrario, el índice *i* no se ajusta si se selecciona el primer subgrupo. La función *word()* devuelve la *i*-ésima palabra en el grupo *n,k* como el valor de palabra descodificada, p. ej., usando la ecuación (13) anterior.

En general, un codificador puede realizar la codificación de longitud variable según la estructura de códigos descrita anteriormente, en donde la estructura de códigos define grupos y subgrupos. Nuevamente, cada uno de los grupos incluye palabras de código que representan valores con los mismos pesos. Las palabras de código en cada uno de los grupos se ordenan lexicográficamente con respecto a los valores representados por las palabras de código. Además, el primer subgrupo en cada grupo incluye palabras de código con una primera longitud y el segundo

subgrupo incluye palabras de código con una segunda longitud, distinta a la primera longitud.

La estructura de códigos puede representarse por una estructura de datos a la que puede acceder un codificador o descodificador para realizar la codificación de longitud variable. Como se ha descrito anteriormente, la estructura de datos puede especificar palabras de código básicas para cada uno de los subgrupos, posiciones de palabras de código dentro de cada uno de los grupos, un cierto número de palabras de código dentro de cada uno de los primeros subgrupos y longitudes de las palabras de código dentro de cada uno de los subgrupos. Esta estructura de datos puede almacenarse en una memoria asociada a uno entre un codificador de vídeo, un codificador de imagen, un codificador de audio, un codificador de voz, un descodificador de vídeo, un descodificador de imagen, un descodificador de audio o un descodificador de voz, y puede ser accesible según se necesite para dar soporte a las operaciones de codificación.

Como se ha descrito anteriormente, un descodificador, tal como la unidad 52 de descodificación de entropía, puede seleccionar, en una búsqueda desde arriba hacia abajo (desde el extremo superior al inferior) del árbol de codificación, seleccionando un primer subgrupo de los subgrupos con una palabra de código básico que sea menor o igual que la palabra de código a descodificar. Luego, el descodificador puede determinar la posición de la palabra de código a descodificar dentro del subgrupo seleccionado, es decir, el índice de subgrupo, en base a una diferencia entre la palabra de código a descodificar y la palabra de código básica para el subgrupo seleccionado. El descodificador determina el peso del valor representado por la palabra de código a descodificar, en base al grupo en el cual reside el subgrupo seleccionado, y determina la posición, es decir, el índice de grupo, de la palabra de código dentro del grupo en el cual reside el subgrupo seleccionado, en base a si el subgrupo seleccionado es el primer subgrupo o el segundo subgrupo para el grupo. El descodificador selecciona luego uno de los valores en base al peso del valor representado por la palabra de código a descodificar y la posición de la palabra de código dentro del grupo en el cual reside el subgrupo seleccionado, y descodifica la palabra de código a descodificar con el valor seleccionado. El valor puede corresponder, por ejemplo, a uno de los códigos de bloque en la TABLA 6.

La estructura de códigos y la estructura de datos contempladas según este aspecto de la presente divulgación puede dar soporte a la eficacia en términos de sobregasto de cálculo, utilización de memoria y tiempo de procesamiento. El ejemplo de proceso de descodificación de la TABLA 8, por ejemplo, requiere entre 1 y S comparaciones y consultas para hallar un subgrupo, una o dos sumas, una operación de desplazamiento, una comparación extra y tres consultas extra. El número de etapas necesarias para hallar un subgrupo puede reducirse adicionalmente colocando palabras de código básicas en un árbol de búsqueda binaria, o usando una tabla de consulta extra, pero en ambos casos, a costa de memoria extra.

Al final del proceso de descodificación, según lo anteriormente descrito, el peso k y el índice $l_{n,k}(w)$ para una palabra de código se convierten en valores efectivos (p. ej., por la función *word()* en la TABLA 8). Si los bloques son razonablemente cortos, esto puede lograrse con una sencilla consulta. En caso contrario, la palabra puede sintetizarse usando una fórmula de enumeración, p. ej., según lo descrito en la obra de D. A. Huffman “A method for the construction of minimum-redundancy codes” [“Un procedimiento para la construcción de códigos de redundancia mínima”], Proc. IRE, 40 (Sept. 1952) 1098-1101. Desde una perspectiva de complejidad, este proceso es similar al cálculo de índices en el codificador.

Usando los procesos de codificación y descodificación descritos anteriormente, puede definirse un sistema para la codificación y descodificación adaptable de bloques de datos. Para este ejemplo, se supone que los bloques de entrada pueden codificarse en las siguientes condiciones:

1. No hay ningún contexto, es decir, se implementa un código universal;

2. El contexto está dado por un bloque previamente observado, es decir, $t = n$; y

3. El contexto está dado por dos bloques previamente observados, es decir, $t = 2n$. En lugar de usar bloques efectivos como contextos, es suficiente (debido a la naturaleza sin memoria del origen) usar sus pesos. Esto significa que, para muestras de t bits, se proporciona una formación de $t + 1$ estructuras de códigos indizadas por sus pesos s. Para ahorrar espacio adicional, puede usarse la simetría de las distribuciones-KT con respecto a s y k. En particular, el proceso puede reemplazar $s = t - s$ y mutar los bits (es decir, forzar $k = n - k$) toda vez que $s > t / 2$. De esta manera, sólo es necesario definir $t / 2 + 1$ tablas. En este ejemplo, la cantidad global de memoria necesaria para el código adaptable pasa a ser $1 + n / 2 + 1 + n + 1 = 1{,}5n + 3$ tablas. Las estimaciones específicas de memoria para los tamaños de bloque $n = 8 \ldots 20$ se muestran en la TABLA 9 a continuación.

TABLA 9

Estimaciones [en octetos] de uso de memoria para distintos tamaños de bloque				
n	máx *t*	máx *S*	Tamaño de una única tabla	Tablas para todos los contextos
8	16	14	102	1530
12	24	19	140	2940
16	32	25	184	4968
20	40	29	216	7128

Las tablas anteriores se generaron usando densidades de estimaciones-KT, y usando un algoritmo modificado de Huffman de construcción de códigos, según la presente divulgación. Se expone a continuación un ejemplo de código de ordenador para un programa que implementa un codificador adaptable de bloques según lo descrito en la presente divulgación.

```
/* bitstream.h: */

typedef struct _BITSTREAM BITSTREAM;

void bitstream_open(BITSTREAM *p, unsigned char *pbs, unsigned bit_offset, int read);
void bitstream_close(BITSTREAM *p, unsigned char **p_pbs, unsigned *p_bit_offset, int write);

void put_bits(unsigned bits, int len, BITSTREAM *p);
unsigned bitstream_buffer(BISTREAM *p);
void scroll_bitstream(int len, BITSTREAM *p);


/* blade.h: */

/* encoder functions: */
void blade_enc_init(void);
unsigned blade_enc_0(unsigned block, BITSTREAM *bs);
unsigned blade_enc_1(unsigned block, unsigned cx, BITSTREAM *bs);
unsigned blade_enc_2(unsigned block, unsigned cx1, unsigned cx2, BITSTREAM *bs);

/* decoder functions: */
void blade_dec_init(void);
unsigned blade_dec_0(unsigned *block, BITSTREAM *bs);
unsigned blade_dec_1(unsigned *block, unsigned cx, BITSTREAM *bs);
unsigned blade_dec_2(unsigned *block, unsigned cx1, unsigned cx2, BITSTREAM *bs);


/* blade_12.c:  implements 12-bit BLADE encoder/decoder */

#define N   12      /* block size */
#define SGS 19      /* max # of subgroups */
```

```
/* encoder structure: */
typedef struct {
  unsigned short  nk [N+1];        /* # of elements in first (n,k) subgroup */
  unsigned char   len [SGS];       /* subgroup -> code length mapping */
  unsigned char   sg [N+1][2];     /* (k,j) -> subgroup index mapping */
  unsigned int    base [SGS];      /* subgroup -> base codeword mapping */
} BLADE_ENC;

/* w -> (k,index) mapping: */
static struct {unsigned short k:4, i:12;} w_ki[1<<N];

/*
 * BLADE encoder:
 * Returns:
 *  # of bits set in encoded pattern
 */
unsigned blade_enc (unsigned w, BLADE_ENC *enc, BITSTREAM *bs)
{
  unsigned i, j, k, len, code;

  k = w_ki[w].k;                   /* split w into (k,index) */
  i = w_ki[w].i;
  if (i >= enc->nk[k]) {           /* find subgroup containing w */
    i -= enc->nk[k];               /* adjust index */
    j = enc->sg[k][1];
  } else
    j = enc->sg[k][0];
  code = enc->base[j] + i;         /* generate code */
  len = enc->len[j];
  put_bits(code, len, bs);

  return k;
}

/* decoder structure: */
typedef struct {
  unsigned int    sgs;             /* number of subgroups */
  unsigned short  nk [N+1];        /* # of elements in first (n,k) subgroup */
  unsigned char   len [SGS];       /* subgroup -> code length mapping */
  struct {unsigned char k:7,j:1;} kj [SGS]; /* subgroup -> (k,j) mapping */
  unsigned int    lj_base [SGS];   /* subgroup -> left-justified codewords */
} BLADE_DEC;

/* (k,index) -> w mapping:*/
static unsigned short *ki_w[N+1], _w[1<<N];

/*
 * BLADE decoder:
 * Returns:
 *  # of bits set in encoded pattern
 */
unsigned blade_dec (unsigned *w, BLADE_DEC *dec, BITSTREAM *bs)
{
  unsigned i, j, k, len, val;

  val = bitstream_buffer(bs);
  for (j=0; j<dec->sgs; j++)       /* find subgroup */
    if (dec->lj_base[j] <= val)
      break;
  len = dec->len[j];
  scroll_bitstream(len, bs);       /* skip decoded bits  */
  i = (val - dec->lj_base[j]) >> (32-len);
  k = dec->kj[j].k;
  j = dec->kj[j].j;
  if (j)                           /* convert to (n,k)-group's index */
    i += dec->nk[k];
  *w = ki_w[k][i];                 /* produce reconstructed block */

  return k;
}
```

```
/*********************************************
* Pre-computed BLADE decoder tables:
*/
static BLADE_DEC dec_t [1+(N/2+1)+(N+1)] = {
  { /* no context/ universal code: */ 15,
  {1,12,66,92,495,792,924,792,495,122,66,12,1}, {3,3,7,7,10,10,11,11,12,12,13,13,14,14,14},
  {{0,0},{12,0},{1,0},{11,0},{2,0},{10,0},{3,0},{9,0},{3,1},{9,1},{4,0},{8,0},{5,0},{6,0},{7,0}},
  {0xE0000000,0xC0000000,0xA8000000,0x90000000,0x7F800000,0x6F000000,0x63800000,0x54400000,
  0x4C400000,0x46200000,0x36A80000,0x27300000,0x1A800000,0x0C600000,0x00000000} },
  { /* (12,0): */ 17,
  {1,9,66,64,495,792,924,792,334,220,66,11,1}, {1,5,6,9,12,13,15,17,19,20,21,22,22,23,23,24,24},
  {{0,0},{1,0},{1,1},{2,0},{3,0},{3,1},{4,0},{5,0},{6,0},{7,0},{8,0},{8,1},{9,0},{10,0},{11,0},{11,1},{12,0}},
  {0x80000000,0x40000000,0x30000000,0x0F000000,0x0B000000,0x06200000,0x02420000,0x00B60000,
  0x00428000,0x00110000,0x00069000,0x00040000,0x00009000,0x00001800,0x00000200,0x00000100,0x00000080} },
  { /* (12,1): */ 16,
  {1,12,17,220,495,792,924,340,495,220,66,10,1}, {2,5,8,9,11,13,15,16,17,18,18,19,19,19,19,20},
  {{0,0},{1,0},{2,0},{2,1},{3,0},{4,0},{5,0},{6,0},{7,0},{7,1},{8,0},{9,0},{10,0},{11,0},{12,0},{11,1}},
  {0xC0000000,0x60000000,0x4F000000,0x36800000,0x1E000000,0x0B880000,0x05580000,0x01BC0000,
  0x01120000,0x00A10000,0x00254000,0x00098000,0x00018000,0x00004000,0x00002000,0x00000000} },
  { /* (12,2): */ 15,
  {1,12,66,211,495,792,924,792,486,220,66,12,1}, {3,6,8,10,11,12,14,15,16,16,17,17,17,17,17},


  {{0,0},{1,0},{2,0},{3,0},{3,1},{4,0},{5,0},{6,0},{7,0},{8,0},{8,1},{9,0},{10,0},{11,0},{12,0}},
  {0xE0000000,0xB0000000,0x6E000000,0x39400000,0x28200000,0x19300000,0x0CD00000,0x05080000,
  0x02800000,0x00A00000,0x00958000,0x00278000,0x00068000,0x00008000,0x00000000} },
  { /* (12,3): */ 16,
  {1,12,30,220,495,792,924,792,19,220,6,12,1}, {4,6,8,8,10,12,13,14,14,14,14,14,14,15,15,15},
  {{0,0},{1,0},{2,0},{2,1},{3,0},{4,0},{5,0},{6,0},{7,0},{8,0},{10,0},{11,0},{13,0},{8,1},{10,1},{9,0}},
  {0xF0000000,0xC0000000,0xA2000000,0x90000000,0x58000000,0x3A100000,0x21500000,0x12E00000,
  0x06800000,0x06340000,0x06100000,0x05E00000,0x05B00000,0x02300000,0x01B80000,0x00000000} },
  { /* (12,4): */ 16,
  {1,12,66,220,495,363,924,792,495,219,66,4,1}, {5,7,8,10,12,12,12,12,13,13,13,13,13,13,14,14},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{5,0},{11,0},{12,0},{5,1},{11,1},{6,0},{7,0},{9,0},{10,0},{9,1},{8,0}},
  {0xF8000000,0xE0000000,0xBF000000,0x88000000,0x69100000,0x56200000,0x55E00000,0x55D00000,
  0x46880000,0x46480000,0x29680000,0x10A80000,0x09D00000,0x07000000,0x07BC0000,0x00000000} },
  { /* (12,5): */ 15,
  {1,12,66,220,495,792,508,792,350,220,66,12,1}, {6,8,10,10,11,11,12,12,12,12,12,12,13,13,13},
  {{0,0},{1,0},{2,0},{12,0},{3,0},{11,0},{4,0},{5,0},{6,0},{8,0},{9,0},{10,0},{6,1},{8,1},{7,0}},
  {0xFC000000,0xF0000000,0xDF800000,0xDF400000,0xC3C00000,0xC2400000,0xA3500000,0x71D00000,
  0x62000000,0x5C200000,0x2E800000,0x2A400000,0x1D480000,0x18000000,0x00000000} },
  { /* (12,6): */ 15,
  {1,12,66,47,495,792,924,792,495,85,66,12,1}, {8,8,9,9,11,11,11,11,12,12,12,12,12,12,13},
  {{0,0},{12,0},{1,0},{11,0},{2,0},{5,0},{9,0},{10,0},{3,1},{9,1},{4,0},{6,0},{7,0},{8,0},{8,0}},
  {0xFF000000,0xFE000000,0xF8000000,0xF2000000,0xE9C00000,0xE3E00000,0xD9400000,0xD1000000,
  0xC6300000,0xBE000000,0x9ED00000,0x6D500000,0x3B000000,0x1CE00000,0x00000000} },
  { /* (24,0): */ 19,
  {1,12,25,220,487,791,924,787,494,220,66,11,1}, {1,5,9,10,13,16,17,19,20,22,24,25,26,27,28,30,31,32,32},
  {{0,0},{1,0},{2,0},{2,1},{3,0},{4,0},{4,1},{5,0},{5,1},{6,0},{7,0},{7,1},{8,0},{8,1},{9,0},{10,0},{11,0},{11,1},{12,0}},
  {0x80000000,0x20000000,0x13800000,0x09400000,0x02600000,0x00780000,0x00750000,0x00122000,0x00121000,
  0x0003A000,0x00005D00,0x00004A80,0x00000F00,0x00000EE0,0x00000120,0x00000018,0x00000002,0x00000001,0x00000000} },
  { /* (24,1): */ 17,
  {1,7,66,220,495,326,924,792,495,4,66,11,1}, {1,5,6,9,12,15,17,18,20,22,23,24,25,26,27,28,28},
  {{0,0},{1,0},{1,1},{2,0},{3,0},{4,0},{5,0},{5,1},{6,0},{7,0},{8,0},{9,0},{9,1},{10,0},{11,0},{11,1},{12,0}},
  {0x80000000,0x48000000,0x34000000,0x13000000,0x05400000,0x01620000,0x00BF0000,0x004A8000,
  0x00100000,0x00046000,0x00009200,0x00007E00,0x00001200,0x00000180,0x00000020,0x00000010,0x00000000} },
```

```
  { /* (24,2): */ 17,
  {1,12,47,220,495,792,924,1,495,220,58,11,1}, {2,5,8,9,11,14,16,18,19,20,21,22,23,24,24,25,25},
  {{0,0},{1,0},{2,0},{2,1},{3,0},{4,0},{5,0},{6,0},{7,0},{7,1},{8,0},{9,0},{10,0},{10,1},{11,0},{11,1},{1
  {0xC0000000,0x60000000,0x31000000,0x27800000,0x0C000000,0x04440000,0x01200000,0x00450000,
  0x0044E000,0x00137000,0x0003F800,0x00008800,0x00001400,0x00000000,0x00000100,0x00000080,0x00000000} },
  { /* (24,3): */ 17,
  {1,6,66,1,495,4,924,792,495,220,66,7,1}, {2,5,6,8,10,11,13,14,15,16,18,19,20,21,21,22,22},
  {{0,0},{1,0},{1,1},{2,0},{3,0},{3,1},{4,0},{5,0},{5,1},{6,0},{7,0},{8,0},{9,0},{10,0},{11,0},{11,1},{12
  {0xC6000000,0x90000000,0x78000000,0x36000000,0x35C00000,0x1A600000,0x0AE80000,0x0AD80000,
  0x04B00000,0x01140000,0x004E0000,0x00102000,0x00026000,0x00005000,0x00001800,0x00000400,0x00000000} },
  { /* (24,4): */ 15,
  {1,12,66,220,495,10,924,792,495,220,66,7,1}, {3,6,8,10,12,13,14,15,16,17,18,19,19,20,20},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{5,0},{5,1},{6,0},{7,0},{8,0},{9,0},{10,0},{11,0},{11,1},{12,0}},
  {0xE0000000,0xB0000000,0x6E000000,0x37000000,0x18100000,0x17C00000,0x0B800000,0x04500000,
  0x01380000,0x00400000,0x00098000,0x00014000,0x00006000,0x00001000,0x00000000} },
  { /* (24,5): */ 16,
  {1,12,66,220,495,792,451,792,2,220,66,11,1}, {4,6,8,10,12,13,14,15,16,16,17,17,18,18,19,19},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{5,0},{6,0},{6,1},{7,0},{8,0},{8,1},{9,0},{10,0},{11,0},{11,1},{12,0}},
  {0xF8000000,0xC0000000,0x7E000000,0x47000000,0x28100000,0x0F500000,0x08440000,0x04920000,
  0x017A0000,0x01780000,0x00818000,0x00138000,0x00030000,0x00004000,0x00002000,0x00000000} },
  { /* (24,6): */ 17,
  {1,8,65,220,2,792,924,792,495,220,59,12,1}, {4,6,7,8,9,10,11,12,13,14,15,16,16,16,17,17,17},
  {{0,0},{1,0},{1,1},{2,0},{2,1},{3,0},{4,0},{4,1},{5,0},{6,0},{7,0},{8,0},{9,0},{10,0},{10,1},{11,0},{12
  {0xFC000000,0xD0000000,0x68000000,0x87000000,0x86B00000,0x4F800000,0x4F400000,0x30700000,
  0x17B00000,0x09400000,0x03100000,0x01210000,0x004E0000,0x000A0000,0x00068000,0x00009000,0x00000000} },
  { /* (24,7): */ 15,
  {1,12,66,220,495,62,924,792,495,220,66,8,1}, {5,7,9,10,11,12,13,13,14,15,15,15,15,15,16},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{5,0},{5,1},{6,0},{7,0},{8,0},{9,0},{10,0},{11,0},{12,0},{11,1}},
  {0xF8000000,0xE0000000,0xBF000000,0x88000000,0x4A200000,0x46400000,0x2F700000,0x12800000,
  0x06300000,0x02520000,0x00980000,0x00160000,0x00060000,0x00040000,0x00000000} },
  { /* (24,8): */ 15,
  {1,12,66,220,287,792,924,792,495,220,62,12,1}, {6,8,9,10,11,12,12,13,14,14,14,14,14,14,15},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{4,1},{5,0},{6,0},{7,0},{8,0},{9,0},{10,0},{11,0},{12,0},{10,1}},
  {0xFC000000,0xF0000000,0xCF000000,0x98000000,0x74200000,0x67200000,0x35A00000,0x18C00000,
  0x0C600000,0x04A40000,0x01340000,0x00300000,0x000C0000,0x00080000,0x00000000} },
  { /* (24,9): */ 14,
  {1,12,66,220,417,792,924,792,495,220,66,12,1}, {7,8,9,11,11,12,12,13,13,13,13,13,13,14},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{4,1},{5,0},{6,0},{7,0},{8,0},{10,0},{11,0},{12,0},{9,0}},
  {0xFE000000,0xF2000000,0xD1000000,0xB5800000,0x91600000,0x7C800000,0x4B000000,0x2E200000,
  0x15600000,0x05E80000,0x03D80000,0x03780000,0x03700000,0x00000000} },
  { /* (24,10): */ 15,
  {1,12,66,220,221,792,923,792,495,220,66,12,1}, {7,9,10,11,11,12,12,12,12,12,12,13,13,13,13},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{4,1},{5,0},{6,0},{10,0},{11,0},{12,0},{6,1},{7,0},{8,0},{9,0}},
  {0xFE000000,0xF8000000,0xE7800000,0xCC000000,0xB0600000,0x9F400000,0x6DC00000,0x34100000,
  0x2FF00000,0x2F300000,0x2F200000,0x2F180000,0x16580000,0x06E00000,0x00000000} },
  { /* (24,11): */ 14,
  {1,12,23,220,495,792,924,792,495,220,66,12,1}, {8,10,10,11,11,11,11,12,12,12,12,12,12,13},
  {{0,0},{1,0},{2,0},{2,1},{3,0},{11,0},{12,0},{4,0},{5,0},{6,0},{8,0},{9,0},{10,0},{7,0}},
  {0xFF000000,0xFC000000,0xF6400000,0xFGE00000,0xD5600000,0xD3E00000,0xB3C00000,0xB4D00000,
  0x83500000,0x49900000,0x2AA00000,0x1CE00000,0x18C00000,0x00000000} },
  { /* (24,12): */ 14,
  {1,12,66,220,495,792,504,792,495,220,66,12,1}, {10,10,10,10,11,11,12,12,12,12,12,12,12,13},
  {{0,0},{1,0},{11,0},{12,0},{2,0},{10,0},{3,0},{4,0},{5,0},{6,0},{7,0},{8,0},{9,0},{6,1}},
  {0xFF000000,0xFC000000,0xF9C00000,0xF9800000,0xF1400000,0xE9C00000,0xDB400000,0xBC500000,
  0x8AD00000,0x6B500000,0x39D00000,0x1AE00000,0x0D200000,0x00000000} }
};


/* encoder tables (computed using decoder's tables): */
static BLADE_ENC enc_t [1+(N/2+1)+(N+1)];
```

```
/* initialize encoder: */
void blade_enc_init()
{
  unsigned int i[N+1], j, k, l, w;
  /* init enc[]: */
  for (j=0; j<1+(N/2+1)+(N+1); j++) {
    for (k=0; k<=N; k++) enc_t[j].nk[k] = dec_t[j].nk[k];
    for (k=0; k<=SGS; k++) {
      enc_t[j].sg[dec_t[j].kj[k].k][dec_t[j].kj[k].j] = j;
      enc_t[j].jen[k] = dec_t[j].jen[k];
      enc_t[j].base[k] = dec_t[j].jj_base[k] >> (32-dec_t[j].jen[k]);
    }
  }
  /* init w_ki[]: */
  for (j=0; k<=N; k++) i[k] = 0;
  for (w=0; w<(1<<N); w++) {
    for (k=0,j=0; j<N; j++) if (w & (1<<j)) k++;
    w_ki[w].k = k;
    w_ki[w].i = i[k];
    i[k] ++;
  }
}

/* initialize decoder: */
void blade_dec_init()
{
  static short b[N+1] = {1,12,66,220,495,792,924,792,495,220,66,12,1};
  unsigned int i[N+1], j, k, w;
  /* init ki_w[]: */
  for (j=0,k=0; k<=N; j+=b[k],k++) {ki_w[k] = _w + j; i[k] = 0;}
  for (w=0; w<(1<<N); w++) {
    for (k=0,j=0; j<N; j++) if (w & (1<<j)) k++;
    ki_w[k][i[k]] = w;
    i[k] ++;
  }
}
```

```
/* encoder's functions: */
unsigned blade_enc_0 (unsigned w, BITSTREAM *bs)
{
  return blade_enc (w, enc_t + 0, bs);
}

unsigned blade_enc_1 (unsigned w, unsigned cx, BITSTREAM *bs)
{
  unsigned r;
  if (cx > N/2)
    r = N - blade_enc (w ^((1<<N)-1), enc_t + 1 + N - cx, bs);
  else
    r = blade_enc (w, enc_t + 1 + cx, bs);
  return r;
}

unsigned blade_enc_2 (unsigned w, unsigned cx1, unsigned cx2, BITSTREAM *bs)
{
  unsigned cx = cx1 + cx2, r;
  if (cx > N)
    r = N - blade_enc (w ^((1<<N)-1), enc_t + 1 + (N/2 + 1) + 2*N - cx, bs);
  else
    r = blade_enc (w, enc_t + 1 + (N/2 + 1) + cx, bs);
  return r;
}

}


/* main.c - test program and demo: */

#define M   1000    /* max # of blocks in test sequence */
#define Q   1000000 /* # of iterations */

/* test program: */
int main ()
{
  /* in/out buffers: */
  static unsigned char in_buffer [M*N/8];
  static unsigned char out_buffer [M*N/8 + 1024];
  static BITSTREAM in, out;

  /* vars: */
  unsigned char *pbs; int bit_offset;
  unsigned int w, cx, cx1 = 0, cx2 = 0;
  int i, j, k, m;
  double p, h, c;
```

```
/* init BLADE-12 library: */
blade_init ();

/* scan sources: */
for (p=0.01; p<=0.991; p+=0.01) {

  /* estimate entropy: */
  h = - (p * log(p) + (1.-p) * log(1.-p)) / log(2.);
  printf ("\np=%g, h=%g\n", p, h);

  /* try different # of blocks: */
  for (m=1; m<N; m++)
  {
    c = 0.;
    /* reset generator: */
    srand(1);
    /* make Q runs: */
    for (i=0; i<Q; i++) {

      /* generate test sequence: */
      memset(in_buffer, 0, sizeof in_buffer);
      bitstream_open(&in, in_buffer, 0, 0);
      for (j=0; j<N*m; j++) {
        /* get a next bit from a pseudo-Bernoulli source: */
        k = ((double) rand() / (double) RAND_MAX) > (1. - p);
        /* insert it in bitstream: */
        put_bits(k, 1, &in);
      }
      bitstream_close (&in, &pbs, &bit_offset, 1);

      /* start encoding: */
      memset(out_buffer, 0, sizeof out_buffer);
      bitstream_open(&out, out_buffer, 0, 0);
      bitstream_open(&in, in_buffer, 0, 1);

      /* run the encoder: */
      for (j=0; j<m; j++) {
        /* block to be encoded: */
        w = (unsigned)get_bits (N, &in);
        /* choose context and encode: */
        if (j == 0)
          cx1 = blade_enc_0 (w, &out);          /* no context */
        else if (j == 1)
          cx2 = blade_enc_1 (w, cx1, &out);     /* use cx1 */
        else {
          cx = blade_enc_2 (w, cx1, cx2, &out); /* use cx1 and cx2 */
          /* scroll contexts: */
          cx1 = cx2;
          cx2 = cx;
        }
      }
      /* close bitstreams: */
      bitstream_close (&in, &pbs, &bit_offset, 0);
      bitstream_close (&out, &pbs, &bit_offset, 1);
```

```
/* compute coding cost: */
c += (double)((pbs - out_buffer) * 8 + bit_offset) / (double)(m*N);

/* start decoding: */
bitstream_open (&in, in_buffer, 0, 1);
bitstream_open (&out, out_buffer, 0, 1);

/* run the decoder: */
for (j=0; j<m; j++) {
  /* choose the context and decode: */
  if (j == 0)
    cx1 = blade_dec_0 (&w, &out);          /* no context */
  else if (j == 1)
    cx2 = blade_dec_1 (&w, cx1, &out);     /* use cx1 */
  else {

            cx = blade_dec_2 (&w, cx1, cx2, &out); /* use cx1 and cx2 */
            /* scroll contexts: */
            cx1 = cx2;
            cx2 = cx;
          }
          /* compare with the original block: */
          if (w != get_bits (N, &in)) {
            printf("?%d,", j);
          }
        }
        /* close bitstreams: */
        bitstream_close (&in, &pbs, &bit_offset, 0);
        bitstream_close (&out, &pbs, &bit_offset, 0);
      }

      /* print results: */
      c /= (double)Q;
      printf("[%d,%g], ", m*N, (c-h)/h);
      fflush(stdout);
    }
  }
  return 1;
}
```

Los resultados experimentales de la evaluación de las prestaciones de un proceso de codificación adaptable, según lo descrito en el presente documento, con tamaño de bloque *n* = 16, se describirán ahora y se compararán con otros algoritmos bien conocidos. En particular, el proceso de codificación adaptable se compara con el algoritmo codificador-Q descrito en la obra de W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, Jr. y R. B. Arps “An overview of the basic principles of the Q-Coder adaptive binary arithmetic coder” [“Un panorama de los principios básicos del codificador aritmético binario adaptable Codificador-Q”], IBM J. Res. Dev., 32 (6) (1988) 717, que se usa en el estándar de codificación de imagen JBIG, y el algoritmo CABAC descrito en la obra de D. Marpe, H. Schwartz y T. Wiegand “Context-Based Adaptive Binary Arithmetic Coding in the H.264 / AVC video compression standard” [“Codificación aritmética binaria adaptable basada en el contexto en el estándar de compresión de vídeo H.264 / AVC”], IEEE Trans. on CSVT, 13(7):620-636, julio de 2003, que se usa en los estándares ITU-T H.264 / MPEG para la compresión de vídeo.

A fin de llevar a cabo las pruebas, se usaron secuencias generadas por ordenador para simular la salida de un

origen binario de Bernoulli con probabilidad p. Las longitudes de tales secuencias variaron entre 16 a 1024 y, para cada longitud específica, se generaron Q = 1.000.000 de muestras de tales secuencias. Las tasas de redundancia relativa se calcularon como:

$$\text{Tasa} = \frac{(\text{suma de las longitudes de todos los códigos producidos para secuencias Q}) / Q - H(p)}{H(p)}$$

Para el proceso de codificación adaptable descrito en la presente divulgación, se usó la siguiente estructura de contextos:

1. el primer bloque de 16 bits se codifica sin contexto (código universal);

2. el segundo bloque se codifica usando el primero como su contexto (código con t = 16); y

3. el tercero, y todos los bloques subsiguientes, se codifican usando dos bloques previos en secuencia como contextos (código basado en muestras con t = 32).

Las FIGS. 7A y 7B son gráficos que ilustran la comparación de tasas de redundancia de un código de bloques adaptable con técnicas de codificador–Q y CABAC, con distintos valores de *p*. La FIG. 7A muestra los resultados para p = 0,1, mientras la FIG. 7B muestra los resultados para p = 0,5. La FIG. 8 es un gráfico que ilustra la sensibilidad de la redundancia a la asimetría de los datos de origen para las técnicas del código de bloques adaptable, el codificador-Q y la CABAC. Los resultados del estudio experimental se muestran en las FIGS. 7A, 7B y 8. Las FIGS. 7A y 7B grafican el número de bits codificados sobre el eje x con respecto a (longitud media de código – Entropía) / Entropía sobre el eje y.

La FIG. 8 grafica la probabilidad en el eje x con respecto a (longitud media de código – Entropía) / Entropía sobre el eje y. Cada gráfico en las FIGS. 7A, 7B y 8 muestra gráficos para el codificador–Q, la CABAC y la codificación adaptable con los rótulos correspondientes. A partir de los resultados experimentales, puede verse que el código adaptable descrito en la presente divulgación puede tener una velocidad mucho más rápida de convergencia que ambos algoritmos del codificador-Q y la CABAC. El proceso de codificación adaptable supera en prestaciones a los algoritmos del codificador-Q y de la CABAC para secuencias más cortas, y se hace comparable con los algoritmos del codificador-Q y la CABAC cuando la longitud total de los bits codificados se aproxima a 1024. Como se muestra adicionalmente en la FIG. 8, después de 160 bits codificados (o bien bloques de 16 bits), el proceso de codificación adaptable puede brindar una menor redundancia en comparación con los algoritmos del codificador-Q y la CABAC. Este comportamiento es coherente con el Teorema 1 expuesto anteriormente.

La FIG. 9 es un diagrama de flujo que ilustra un procedimiento para construir una codificación de longitud variable, eficiente en términos de memoria, para distribuciones monótonas, según un primer aspecto general de la presente divulgación. El procedimiento puede ser implementado por un procesador asociado a un codificador, descodificador u otro dispositivo, a fin de construir códigos para su uso por parte de una unidad 46 de codificación de entropía y una unidad 52 de descodificación de entropía, según se muestra en las FIGS. 2 y 3, y puede dar soporte a la compresión y la codificación de cualquiera entre una gran variedad de datos, que incluyen, sin limitarse a, los datos de vídeo, imagen, voz y audio. Tal procesador puede proporcionarse, por ejemplo, dentro de un codificador o descodificador, o dentro de un sistema informático de propósito general, p. ej., para preparar una estructura de datos que define atributos de estructuras de códigos útiles en la codificación de longitud variable.

Como se muestra en la FIG. 9, el procesador obtiene un alfabeto de símbolos de entrada a codificar (70). Los símbolos tienen respectivos pesos, que indican la probabilidad o frecuencia de presencia o uso de los símbolos en un conjunto o secuencia dada de datos. Al determinar los pesos (72) pertinentes de símbolos, el proceso asigna índices a los símbolos en base a sus pesos (74) y asigna códigos a los símbolos en base a los índices y al orden lexicográfico de los símbolos (76). Por tanto, los símbolos con los mismos pesos pueden ordenarse lexicográficamente para facilitar las técnicas de codificación, según lo descrito en la presente divulgación.

Los códigos pueden organizarse según una estructura de códigos representada por un árbol de codificación binaria. El procesador identifica una palabra de código básica para cada nivel en el árbol (78) de codificación. La palabra de código básica es la más pequeña palabra de código en un nivel dado en el árbol, y corresponde al símbolo lexicográficamente más anterior entre los símbolos en ese nivel en el árbol. Para proporcionar una estructura compacta de datos, el procesador quita un número fijo B de los bits iniciales de las palabras de código básicas, a fin de producir las palabras (80) de código básicas parciales. Las palabras de código básicas pueden formularse como palabras de código alineadas a izquierda, y los bits iniciales pueden ser los M bits iniciales, yendo de derecha a izquierda, en las palabras de código alineadas a izquierda. En algunas implementaciones, el número de bits iniciales que se quitan puede ser 8. En otras implementaciones, el número de bits iniciales que se quitan puede ser menor o mayor que 8.

Para muchos niveles del árbol de codificación, los M bits iniciales serán ceros. En algunos niveles, sin embargo, los bits iniciales pueden formar todo, o parte de, el código básico para el respectivo nivel en el árbol. En estos niveles seleccionados, el procesador genera indicadores (82) de omisión. Los indicadores de omisión proporcionan una instrucción para un descodificador, a fin de desplazar el flujo de bits en B bits, de modo que el código básico no se pierda al quitar los B bits iniciales. El proceso puede almacenar, en una estructura de datos, las palabras de código básicas parciales resultantes, las longitudes asociadas a las palabras de código en los respectivos niveles del árbol de codificación, los desplazamientos que indican los índices de los respectivos símbolos asociados a las palabras de código en el árbol de codificación y uno o más indicadores de omisión que indican cuándo debería desplazarse el flujo de bits en B bits para impedir la pérdida de una palabra de código básica que caiga al menos parcialmente dentro de los B bits iniciales (84). La estructura de datos puede proporcionarse a la unidad 46 de codificación de entropía y a la unidad 52 de descodificación de entropía, para asistir en la realización de las operaciones de codificación y descodificación de entropía con los códigos variables construidos. La estructura de datos puede tomar una gran variedad de formas, incluyendo una o más tablas de consulta unidimensionales o multidimensionales, árboles binarios, árboles radicales, ficheros planos o similares.

La FIG. 10 es un diagrama de flujo que ilustra un procedimiento para codificar símbolos usando códigos de longitud variable construidos según el procedimiento de la FIG. 9, según la presente divulgación. Como se muestra en la FIG. 10, la unidad 46 de codificación de entropía recibe un símbolo (86), determina un índice para el símbolo (87) y compara el índice de símbolo con una tabla de desplazamientos para identificar un correspondiente nivel en el árbol de codificación. En particular, la unidad 46 de codificación de entropía determina si el índice de símbolo es mayor o igual que el desplazamiento para un nivel dado del árbol (88). El índice de símbolo representa el orden del símbolo entre los otros símbolos, clasificados por orden de peso, estando los símbolos del mismo peso ordenados lexicográficamente, en coherencia con el alfabeto de símbolos. El desplazamiento es la diferencia entre la longitud del código, o códigos, para el nivel pertinente del árbol, y la máxima longitud de código. En el árbol de la FIG. 4, si la máxima longitud de código es 16, por ejemplo, y la longitud de código en el nivel 3 del árbol es 3, entonces el desplazamiento pertinente para la palabra de código básica es 12.

Si el índice de símbolo no supera al desplazamiento para el nivel actual del árbol, la unidad 46 de codificación de entropía prosigue hacia abajo hasta el siguiente nivel (90) del árbol de codificación en una búsqueda de arriba a abajo y repite la comparación del índice de símbolo con el desplazamiento para ese siguiente nivel (88). Cuando la unidad 46 de codificación de entropía determina que el índice de símbolo es mayor o igual que el desplazamiento para un nivel específico del árbol (88) de codificación, la unidad de codificación de entropía calcula la palabra de código adecuada para el símbolo. En particular, la unidad 46 de codificación de entropía fija la palabra de código para el símbolo como la suma de la palabra de código básica para el nivel actual del árbol más la diferencia entre el índice de símbolo y el desplazamiento para ese nivel (92).

Usando el ejemplo de árbol de la FIG. 4, si el índice de símbolo es 14, la unidad 46 de codificación de entropía determina que el código para el símbolo reside en el nivel 3 del árbol porque 14 es mayor que el desplazamiento de 12 asignado a la palabra de código básica para ese nivel. La unidad 46 de codificación de entropía calcula luego la palabra de código como la palabra de código básica (001) + (el índice de símbolo (14) – el desplazamiento (12)), es decir, 001 + 2 = 001 + 010 = 011. Al fijar el código para el símbolo recibido (92), la unidad 46 de codificación de entropía emite la palabra de código al flujo (94) de bits para su transmisión, p. ej., a un dispositivo receptor con la unidad 52 de descodificación de entropía. La unidad 46 de codificación de entropía repite luego el proceso para el próximo símbolo en la secuencia pertinente de datos.

La FIG. 11 es un diagrama en bloques que ilustra un procedimiento para descodificar códigos de longitud variable construidos según el procedimiento de la FIG. 9, según la presente divulgación. El procedimiento mostrado en la FIG. 11 puede llevarse a cabo usando un algoritmo idéntico o similar al mostrado en la TABLA 5. En un ejemplo de implementación, los códigos pueden ser recibidos por la unidad 52 de descodificación de entropía, y codificados por una unidad 46 de codificación de entropía, según lo descrito con referencia a la FIG. 10. El procedimiento ilustrado en la FIG. 11 puede hacer uso de técnicas de descodificación eficientes en términos de memoria, según lo descrito en la presente divulgación, y puede aprovechar la estructura compacta de datos con la cual pueden construirse tales códigos. Como se muestra en la FIG. 11, la unidad 52 de descodificación de entropía recibe una palabra de código desde un flujo (96) de bits entrante. La palabra de código puede obtenerse a partir de un ancho fijo W de bits recuperados desde un almacén temporal del flujo de bits. La palabra de código puede alinearse a izquierda o convertirse a un formato alineado a izquierda, y el ancho W puede reducirse quitando B bits iniciales de la palabra de código, de derecha a izquierda.

La unidad 52 de descodificación de entropía compara la palabra de código con las palabras de código básicas para distintos niveles de un árbol de codificación, a partir del nivel supremo y avanzando hacia el interior del árbol en una búsqueda de arriba a abajo, hasta que se halle una palabra de código básica adecuada. En particular, la unidad 52 de descodificación de entropía puede determinar si la palabra de código básica para el nivel actual del árbol es menor o igual que la palabra (98) de código. Si no es así, la unidad de descodificación de entropía continúa hacia

abajo hasta el próximo nivel del árbol (100) y repite la comparación (98) para la palabra de código básica asociada al próximo nivel. Al proseguir en el próximo nivel (100), sin embargo, la unidad 52 de descodificación de entropía determina si una indicación (102) de omisión está asociada al nivel actual. Si es así, la unidad 52 de descodificación de entropía desplaza el almacén temporal del flujo de bits en un incremento fijo (104) antes de proseguir hasta el próximo nivel (100). En particular, la unidad 52 de descodificación de entropía puede desplazar el almacén temporal del flujo de bits en M bits, de modo que la palabra de código no se pierda al descartar los M bits iniciales. Si no hay ninguna indicación (102) de omisión, la unidad 52 de descodificación de entropía avanza simplemente hasta el próximo nivel (100).

En cualquier caso, la unidad 52 de descodificación de entropía compara nuevamente la palabra de código con la palabra de código básica para el nivel actual (98). Cuando la unidad 52 de descodificación de entropía halla un nivel en el cual la palabra de código básica es menor o igual que la palabra (98) de código, la unidad 52 de descodificación de entropía determina la longitud residual de las palabras de código básicas en el respectivo nivel (106) y desplaza el flujo de bits en la longitud residual (108). La unidad 52 de descodificación de entropía calcula entonces el símbolo asociado a la palabra (110) de código en base al desplazamiento para el nivel, y la diferencia entre la palabra de código básica y la palabra de código que se está descodificando.

Con referencia al árbol de la FIG. 4, por ejemplo, si la palabra de código es 0110000000000000, entonces la palabra de código truncada y parcial con 8 bits iniciales descartados es 01100000. En este caso, la unidad 52 de descodificación de entropía identificará la palabra de código básica parcial en el nivel 3 (00100000) como menor o igual que la palabra de código, e identificará una longitud residual de 3. La unidad 52 de descodificación de entropía desplaza el flujo 3 de bits hacia adelante para recibir la próxima palabra de código. Además, la unidad 52 de descodificación de entropía calcula el símbolo para la palabra de código sumando el desplazamiento para el nivel 3 a la diferencia entre la palabra de código en el almacén temporal del flujo de bits y la palabra de código básica para el nivel. Por ejemplo, la unidad 52 de descodificación de entropía calcula el código como desplazamiento[3] = 12 más palabra de código 01100000 menos 00100000, que es equivalente a 12 más 2 = 14. En este caso, 14 es el índice del símbolo representado por el código 011.

El procedimiento ilustrado en la FIG. 11 puede aprovechar una estructura de datos muy compacta y una significativa eficiencia de memoria, según se describe en otra parte en la presente divulgación. Como resultado, implementando tal procedimiento, la unidad 52 de descodificación de entropía puede presentar una eficacia aumentada, que incluye un sobregasto de procesamiento reducido, una utilización reducida de memoria y una velocidad de procesamiento aumentada, todo lo cual puede ser deseable para un dispositivo de descodificación de vídeo, u otros dispositivos configurados para la descompresión y la descodificación de datos.

La FIG. 12 es un diagrama de flujo que ilustra un procedimiento para construir códigos de bloque adaptables. El procedimiento de la FIG. 12 puede implementarse dentro de un procesador dentro de un dispositivo de codificación o un procesador de propósito general para dar soporte a la construcción eficaz y directa de códigos de bloques adaptables. Como se muestra en la FIG. 12, el procesador obtiene un conjunto de palabras (112) a codificar. Una estructura de datos que representa la resultante estructura de código puede almacenarse en memoria dentro de un dispositivo de codificación, un dispositivo de descodificación, o ambos. Las palabras pueden ser bloques de códigos binarios. Al determinar los pesos de las palabras (114), el procesador asigna grupos de palabras de código a las palabras en base a los pesos (116). Los grupos de palabras de código incluyen palabras de código para palabras del mismo peso k y pueden abarcar dos niveles adyacentes de un árbol de codificación, p. ej., según se muestra en la FIG. 6.

Como se muestra además en la FIG. 12, el procesador asigna subgrupos a palabras en los mismos grupos, en base a las longitudes de las palabras (118). En particular, un grupo puede incluir un primer subgrupo y un segundo subgrupo. El primer subgrupo contiene una o más palabras de código con la misma longitud y el mismo peso. Análogamente, el segundo subgrupo contiene una o más palabras de código con la misma longitud y el mismo peso. Sin embargo, la longitud de las palabras de código en el primer subgrupo es menor que la longitud de las palabras de código en el segundo subgrupo. Por tanto, cada subgrupo incluye palabras de código del mismo peso y en el mismo nivel en el árbol de codificación.

El procesador identifica una palabra de código básica para cada subgrupo (120). La palabra de código básica es la más pequeña palabra de código en un subgrupo. En el ejemplo de la FIG. 6, la palabra de código básica para el subgrupo 68A es 001. Sin embargo, el subgrupo 68A incluye adicionalmente las palabras de código 010 y 011, además de la palabra de código básica 001. En este ejemplo, las palabras de código en un subgrupo están ordenadas en términos del orden lexicográfico de las palabras que representan, de modo tal que los códigos puedan calcularse inmediata y directamente dada una magnitud de cálculo relativamente pequeña.

El número de elementos en el primer subgrupo de cada grupo puede usarse para calcular las palabras de código. Con ese fin, el procesador almacena el número de elementos en el primer subgrupo de cada grupo (122), y también almacena una asociación (124) de índices de grupo, una asociación (126) de longitudes de código de subgrupo y

una asociación (128) de palabras de código básicas de subgrupo. La asociación de índices de grupo puede identificar la posición de una palabra de código para una palabra en un grupo en el cual resida la palabra de código. La asociación de longitudes de código de subgrupo puede identificar la longitud de los códigos dentro un subgrupo específico. La asociación de palabras de código básicas de subgrupo puede identificar la palabra de código básica asociada a cada subgrupo. En general, un código puede construirse a partir de la palabra de código básica para un subgrupo específico, dado el índice de la palabra dentro del grupo. La información almacenada puede almacenarse en una estructura de datos a la que pueden acceder un codificador, un descodificador, o ambos.

La FIG. 13 es un diagrama de flujo que ilustra un procedimiento para codificar bloques usando códigos de longitud variable construidos según el procedimiento de la FIG. 12. El procedimiento de la FIG. 13 puede implementarse, por ejemplo, dentro de una unidad de codificación de entropía tal como la unidad 46 de codificación de entropía de la FIG. 2. El procedimiento mostrado en la FIG. 13 puede llevarse a cabo usando un algoritmo idéntico o similar al mostrado en la TABLA 7. Como se muestra en la FIG. 13, para codificar una palabra dada, la unidad 46 de codificación de entropía obtiene su peso (130) y un índice (132) de grupo. Usando el peso de la palabra, la unidad 46 de codificación de entropía determina el índice de grupo para la palabra (132) dentro de un árbol de codificación aplicable. El índice $I_{n,k}(w)$ de grupo especifica el índice (posición) de una palabra w en un grupo $W_{n,k}$, que almacena palabras en un orden lexicográfico.

La unidad 46 de codificación de entropía compara el índice de grupo con el número de elementos n_k en el primer subgrupo del grupo en el cual reside la palabra de código para la palabra de entrada. En particular, la unidad 46 de codificación de entropía determina si el índice de grupo es mayor o igual que el número de elementos en el primer subgrupo (134). Si es así, la unidad 46 de codificación de entropía selecciona el segundo subgrupo (138), es decir, el subgrupo 1, en el grupo, y decrementa el valor (140) del índice de grupo. En particular, el valor del índice de grupo se decrementa en el número de elementos en el primer subgrupo n_k. Si el índice de grupo no es mayor o igual que el número de elementos en el primer subgrupo (134), la unidad 46 de codificación de entropía selecciona el primer subgrupo (136), es decir, el subgrupo 0. Cada subgrupo tiene su propia palabra de código básica. La unidad 46 de codificación de entropía obtiene la palabra de código básica para el subgrupo (142) seleccionado y calcula la palabra de código en base a la suma de la palabra de código básica y el valor (144) del índice de grupo.

Con referencia al ejemplo del árbol de codificación de la FIG. 6, como ilustración, si se supone que el peso de la palabra a codificar es 2, el valor del índice de grupo es 2 y el número de elementos en el primer subgrupo es 3, p. ej., correspondientes al grupo en los niveles 3 y 4 del árbol de codificación. En este caso, la unidad 46 de codificación de entropía selecciona el primer subgrupo (subgrupo 0) porque el valor (2) del índice de grupo es menor que el número de elementos (3) en el primer subgrupo. Como resultado, la palabra de código básica es 001. Para codificar la palabra, la unidad 46 de codificación de entropía suma el valor 2 del índice del grupo a la palabra de código básica 001, dando como resultado una palabra de código 011.

Para el mismo grupo, si el valor del índice de grupo fuera 3, la unidad 46 de codificación de entropía seleccionaría el segundo subgrupo (subgrupo 1). Sin embargo, la unidad 46 de codificación de entropía decrementaría el valor del índice de grupo en el número n_k de elementos en el primer subgrupo (subgrupo 0). En este caso, el valor 3 del índice de grupo se reduciría en 3, a cero, y la palabra de código se calcularía como la palabra de código básica 0001 para el segundo subgrupo, más el valor 0 del índice de grupo, dando como resultado una palabra de código 0001.

Además de calcular la palabra de código para la palabra de entrada, la unidad 46 de codificación de entropía puede obtener la longitud de los códigos en el subgrupo seleccionado (146). En el ejemplo anterior, para el subgrupo 0 en el nivel 3, la longitud de los códigos sería 3. La unidad de codificación de entropía emite la palabra de código calculada y la longitud del código de subgrupo al flujo de bits para su almacenamiento y / o transmisión a otro dispositivo, tal como un dispositivo de descodificación que incluye una unidad de descodificación de entropía tal como la unidad 52 de descodificación de entropía.

La FIG. 14 es un diagrama en bloques que ilustra un procedimiento para descodificar códigos de longitud variable construidos según los procedimientos de la FIG. 12 y la FIG. 13. El procedimiento mostrado en la FIG. 14 puede llevarse a cabo usando un algoritmo idéntico o similar al mostrado en la TABLA 8. Los códigos de longitud variable pueden recibirse desde un dispositivo de codificación tal como un dispositivo de codificación que incluye la unidad 46 de codificación de entropía. Los códigos de longitud variable pueden ser recibidos y descodificados por la unidad 52 de descodificación de entropía. Como se muestra en la FIG. 14, la unidad 52 de descodificación de entropía recibe una palabra de código desde el flujo (150) de bits entrante y compara la palabra de código con una palabra de código básica de subgrupo. En particular, la unidad 52 de descodificación de entropía puede buscar en un árbol de codificación aplicable para identificar una palabra de código básica de subgrupo alineada a izquierda que sea menor o igual que el contenido de la palabra de código obtenida desde un almacén temporal (152) del flujo de bits.

Si la palabra de código básica de subgrupo en un subgrupo a un nivel dado en el árbol no es menor o igual que la palabra (152) de código, entonces la unidad 52 de descodificación de entropía prosigue hasta el próximo subgrupo

en el próximo nivel (154) en el árbol y repite la comparación. Este proceso continúa de modo iterativo mientras la palabra de código básica sea mayor que la palabra de código recibida desde el flujo de bits, es decir, hasta que la unidad 52 de descodificación de entropía llegue a un nivel en el cual la palabra de código básica de subgrupo sea menor o igual que la palabra de código. Al comparar la palabra de código y las palabras de código básicas, la unidad 52 de descodificación de entropía puede usar valores incrementales parciales de las palabras de código básicas, para una reducción adicional de memoria, según el primer aspecto de la presente divulgación. En particular, un cierto número de bits iniciales puede descartarse para mejorar la eficacia en términos de memoria, según se ha descrito anteriormente.

Cuando llega a un nivel del árbol de codificación en el cual una palabra de código básica de subgrupo es menor o igual que la palabra de código, la unidad 52 de descodificación de entropía determina la longitud de código para el subgrupo (156) y desplaza el flujo de bits en esa longitud para omitir los bits descodificados y aislar la palabra de código. La unidad 52 de descodificación de entropía puede determinar la posición de la palabra de código en el subgrupo usando la palabra (158) de código básica. Por ejemplo, la unidad 52 de descodificación de entropía puede restar la palabra de código del flujo de bits de la palabra de código básica para producir la diferencia de posición entre la palabra de código y la palabra de código básica.

Como ejemplo, con referencia al árbol de codificación de la FIG. 6, si la palabra de código entrante es 0110000000000000, la unidad 52 de descodificación de entropía identificará la palabra de código básica 0010000000000000 como la palabra de código básica del subgrupo más superior que es menor o igual a la palabra de código. Esta palabra de código básica está asociada al primer subgrupo en el grupo a los niveles 3 y 4. Al determinar la longitud (3) de código del subgrupo asociado a la palabra de código básica, la unidad 52 de descodificación de entropía puede desplazar el flujo de bits para omitir los bits descodificados. La unidad 52 de descodificación de entropía puede determinar el índice de grupo de la palabra de código restando la palabra de código básica de la palabra de código en el flujo de bits. En este ejemplo, 011 menos 001 es igual a 010, lo que produce un valor de 2 del índice de grupo.

La unidad 52 de descodificación de entropía también puede determinar el peso de la palabra (160) de código, p. ej, en base al nivel del subgrupo dentro del árbol de codificación. Además, la unidad 52 de descodificación de entropía puede determinar el índice (162) de subgrupo, es decir, el índice del subgrupo seleccionado dentro del árbol de codificación. Usando el peso, la posición y el índice de subgrupo, la unidad 52 de descodificación de entropía determina el índice (164) de palabra, descodificando por ello la palabra de código del flujo de bits para producir la palabra representada por la palabra de código. Nuevamente, en algunas implementaciones, el procedimiento de descodificación efectuado por la unidad 52 de descodificación de entropía puede corresponder al proceso mostrado en la TABLA 8.

Los expertos en la técnica entenderán que la información y las señales pueden representarse usando cualquiera entre una gran variedad de distintas tecnologías y técnicas. Por ejemplo, los datos, instrucciones, comandos, la información, las señales, bits, símbolos y chips que pueden mencionarse a lo largo de la descripción anterior pueden representarse por voltajes, corrientes, ondas electromagnéticas, campos o partículas magnéticas, campos o partículas ópticas, o cualquier combinación de los mismos.

Aquellos expertos en la técnica apreciarán adicionalmente que los diversos bloques lógicos, módulos, circuitos y etapas de algoritmos ilustrativos descritos con respecto a las realizaciones reveladas en el presente documento pueden implementarse como hardware electrónico, software de ordenador, o combinaciones de ambos. Para ilustrar claramente esta intercambiabilidad de hardware y software, diversos componentes, bloques, módulos, circuitos y etapas ilustrativos han sido descritos anteriormente, en general, en términos de su funcionalidad. Si tal funcionalidad se implementa como hardware o software depende de la aplicación específica y de las restricciones de diseño impuestas sobre el sistema global. Los artesanos expertos pueden implementar la funcionalidad descrita de manera variables para cada aplicación específica, pero tales decisiones de implementación no deberían interpretarse como causantes de un alejamiento del alcance de la presente presente divulgación.

Las técnicas descritas en el presente documento pueden implementarse en hardware, software, firmware o cualquier combinación de los mismos. Tales técnicas pueden implementarse en cualquiera entre una gran variedad de dispositivos tales como ordenadores de propósito general, equipos de mano de comunicación inalámbrica o dispositivos de circuitos integrados con múltiples usos, incluyendo su aplicación en dispositivos de mano de comunicación inalámbrica y otros dispositivos. Cualquier característica descrita como módulos o componentes puede implementarse conjuntamente en un dispositivo de lógica integrada, o por separado como dispositivos lógicos discretos pero interoperables. Si se implementan en software, las técnicas pueden realizarse, al menos en parte, por un medio de almacenamiento de datos legible por ordenador, que comprende código de programa que incluye instrucciones que, cuando se ejecutan, realizan uno o más de los procedimientos descritos anteriormente. El medio de almacenamiento de datos legible por ordenador puede formar parte de un producto de programa de ordenador. El medio legible por ordenador puede comprender memoria de acceso aleatorio (RAM) tal como la

memoria dinámica sincrónica de acceso aleatorio (SDRAM), memoria de sólo lectura (ROM), memoria no volátil de acceso aleatorio (NVRAM), memoria borrable y programable de sólo lectura (EEPROM), memoria FLASH, medios magnéticos u ópticos de almacenamiento de datos, y similares. Las técnicas, adicional o alternativamente, pueden realizarse, al menos en parte, en un medio de comunicación legible por ordenador que lleve o comunique código de programa en forma de instrucciones o estructuras de datos y que puede ser accesible, leído y / o ejecutado por un ordenador, tal como señales u ondas propagadas.

El código de programa puede ser ejecutado por uno o más procesadores, tales como uno o más procesadores de señales digitales (DSP), microprocesadores de propósito general, circuitos integrados específicos de una aplicación (ASIC), formaciones lógicas programables en el terreno (FPGA), u otros circuitos lógicos integrados o discretos equivalentes. Un procesador de propósito general puede ser un microprocesador pero, como alternativa, el procesador puede ser cualquier procesador, controlador, microcontrolador o máquina de estados convencional. Un procesador también puede implementarse como una combinación de dispositivos de cálculo, p. ej., una combinación de un DSP y un microprocesador, una pluralidad de microprocesadores, uno o más microprocesadores conjuntamente con un núcleo de DSP, o cualquier otra configuración tal. En consecuencia, el término “procesador”, según se usa en el presente documento, puede referirse a cualquiera de las estructuras precedentes, o a cualquier otra estructura adecuada para la implementación de las técnicas descritas en el presente documento. Además, en algunos aspectos, la funcionalidad descrita en el presente documento puede proporcionarse con módulos de software dedicados o módulos de hardware configurados para codificar y descodificar, o incorporados en un codificador-descodificador (CODEC) combinado de vídeo.

Se han descrito diversas realizaciones de la presente divulgación. Estas y otras realizaciones están dentro del alcance de las siguientes reivindicaciones.

REIVINDICACIONES

1. Un procedimiento que comprende:

generar (80) valores parciales de palabras de código básicas, correspondiendo las palabras de código básicas a niveles de un árbol (78) de codificación que especifica palabras de código canónicas de longitud variable, estando los valores parciales basados en un desplazamiento de las palabras de código básicas en un cierto número de bits;

generar (82) un indicador de omisión que instruye a un descodificador para omitir un cierto número de bits en un flujo de bits a descodificar antes de avanzar hasta un nivel seleccionado del árbol (78) de descodificación; y

almacenar (84) los valores parciales y el indicador de omisión en una estructura de datos en una memoria.

2. El procedimiento de la reivindicación 1, que comprende adicionalmente:

generar valores representados por las palabras de código básicas;

generar longitudes de los valores parciales de las palabras de código básicas, y

almacenar los valores y las longitudes en la estructura de datos en la memoria.

3. El procedimiento de la reivindicación 1, que comprende adicionalmente almacenar (84) la estructura de datos en una memoria de uno entre un descodificador de vídeo, un descodificador de imagen, un descodificador de audio o un descodificador de voz.

4. El procedimiento de la reivindicación 1, en el cual algunos de los niveles del árbol de codificación incluyen palabras de código dispuestas en orden lexicográfico con respecto a los valores representados por las palabras de código, y cada una de las palabras de código básicas es una palabra de código lexicográficamente mínima en un nivel correspondiente en el árbol (78) de codificación.

5. El procedimiento de la reivindicación 1, en el cual los valores parciales de las palabras de código básicas representan la eliminación de un número fijo de bits iniciales de las palabras de código básicas.

6. El procedimiento de la reivindicación 1, que comprende adicionalmente descodificar una de las palabras de código del flujo de bits de palabras de código, usando la estructura de datos almacenada, en el cual la descodificación comprende:

buscar (98) en los niveles del árbol (78) de codificación un valor seleccionado de los niveles parciales de las palabras de código básicas que sea menor o igual a la palabra de código del flujo de bits de palabras de código;

omitir (108) un cierto número de bits en el flujo de bits de palabras de código antes de avanzar al nivel seleccionado del árbol (78) de codificación en respuesta al indicador de omisión;

calcular uno entre una pluralidad de valores correspondientes a la palabra de código, en base a una diferencia entre el valor seleccionado de los valores parciales de las palabras de código básicas que sea menor o igual que la palabra de código y la palabra de código, y un índice del valor seleccionado de los valores parciales de las palabras de código básicas que sea menor o igual que la palabra de código.

7. El procedimiento de la reivindicación 1, que comprende adicionalmente codificar valores con las palabras de código coherentes con el árbol (78) de codificación, en el cual los valores representados por las palabras de código representan al menos uno entre datos de vídeo, datos de imagen, datos de voz o datos de audio.

8. Un procedimiento de descodificación que comprende:

acceder a una estructura de datos almacenada en memoria, en el cual la estructura de datos comprende valores parciales de palabras de código básicas para niveles de un árbol (78) de codificación que especifican palabras de código canónicas de longitud variable, estando los valores parciales basados en un desplazamiento de las palabras de código básicas en un cierto número de bits, y un indicador de omisión que instruye a un descodificador para omitir un cierto número de bits en un flujo de bits a descodificar antes de proseguir hasta un nivel seleccionado del árbol de codificación; y descodificar una de las palabras de código del flujo de bits, en base a los valores parciales y el indicador de omisión en la estructura de datos almacenada.

9. Un programa de ordenador que comprende instrucciones para causar que un procesador lleve a cabo el procedimiento según una cualquiera de las reivindicaciones 1 a 7.

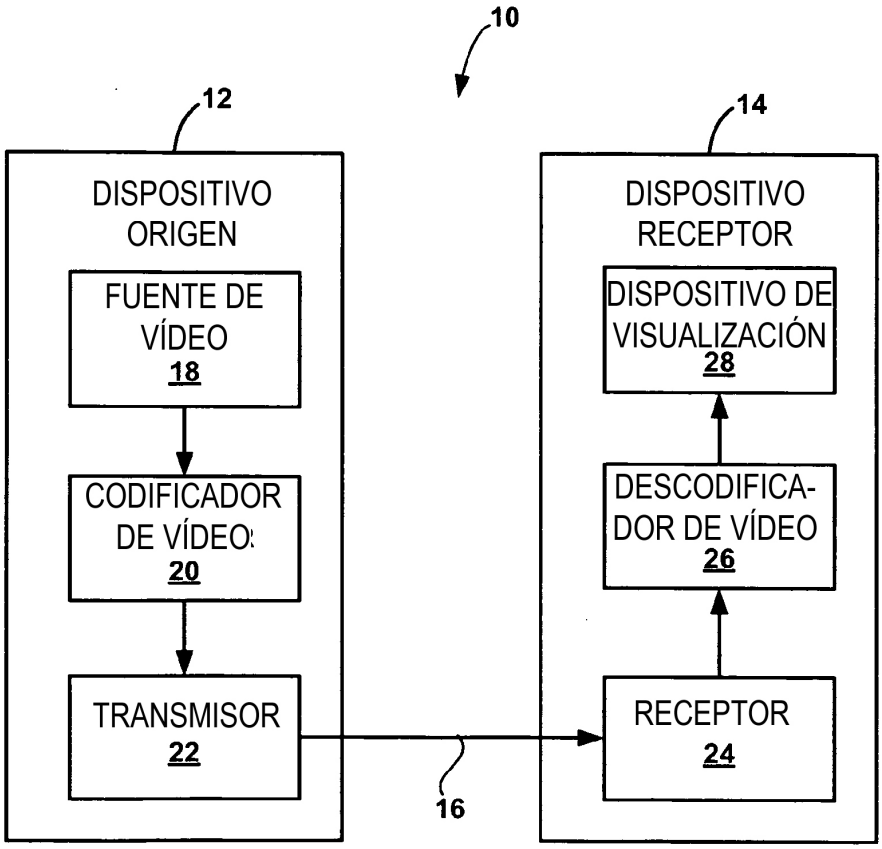
10
12
14
DISPOSITIVO ORIGEN
FUENTE DE VÍDEO
18
CODIFICADOR DE VÍDEO
20
TRANSMISOR
22
16
DISPOSITIVO RECEPTOR
DISPOSITIVO DE VISUALIZACIÓN
28
DESCODIFICA-DOR DE VÍDEO
26
RECEPTOR
24

FIG. 1

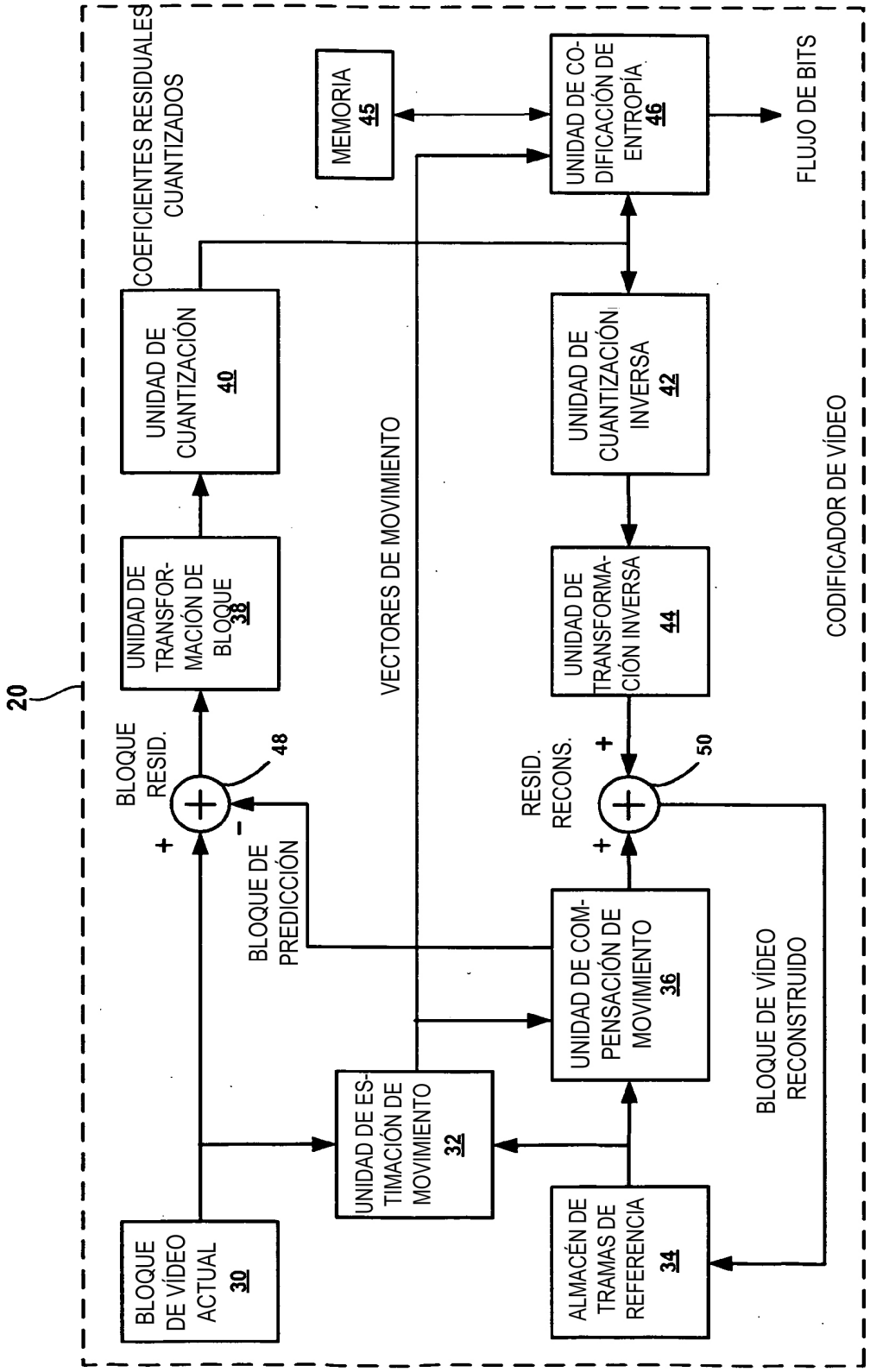
20
CODIFICADOR DE VÍDEO
BLOQUE DE VÍDEO ACTUAL 30
UNIDAD DE ESTIMACIÓN DE MOVIMIENTO 32
ALMACÉN DE TRAMAS DE REFERENCIA 34
UNIDAD DE COMPENSACIÓN DE MOVIMIENTO 36
BLOQUE DE PREDICCIÓN
BLOQUE RESID.
UNIDAD DE TRANSFORMACIÓN DE BLOQUE 38
UNIDAD DE CUANTIZACIÓN 40
COEFICIENTES RESIDUALES CUANTIZADOS
VECTORES DE MOVIMIENTO
MEMORIA 45
UNIDAD DE CODIFICACIÓN DE ENTROPÍA 46
FLUJO DE BITS
UNIDAD DE CUANTIZACIÓN INVERSA 42
UNIDAD DE TRANSFORMACIÓN INVERSA 44
RESID. RECONS.
BLOQUE DE VÍDEO RECONSTRUIDO
48
50

FIG. 2

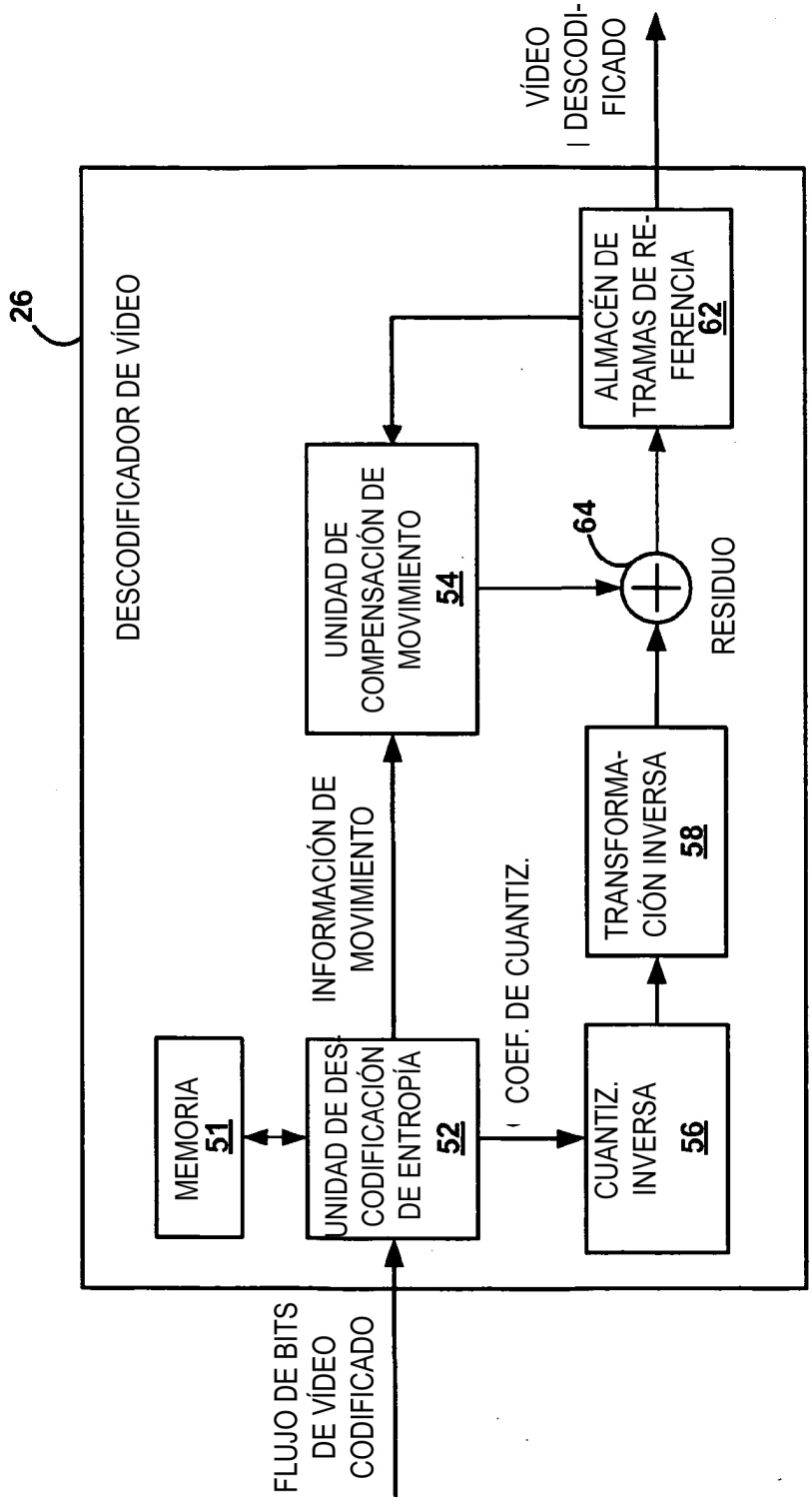
26
DESCODIFICADOR DE VÍDEO
MEMORIA 51
UNIDAD DE DES-CODIFICACIÓN DE ENTROPÍA 52
UNIDAD DE COMPENSACIÓN DE MOVIMIENTO 54
CUANTIZ. INVERSA 56
TRANSFORMA-CIÓN INVERSA 58
ALMACÉN DE TRAMAS DE RE-FERENCIA 62
64
FLUJO DE BITS DE VÍDEO CODIFICADO
INFORMACIÓN DE MOVIMIENTO
COEF. DE CUANTIZ.
RESIDUO
VÍDEO DESCODI-FICADO

FIG. 3

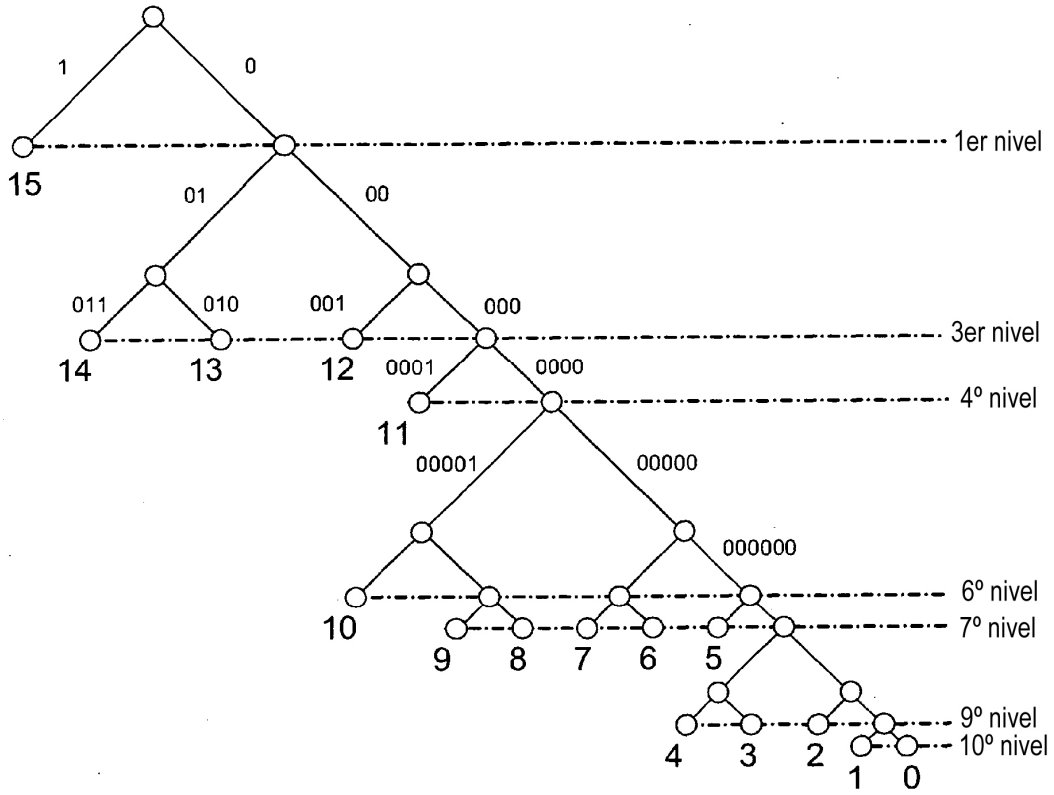
1
0
15
1er nivel
01
00
011
010
001
000
14
13
12
3er nivel
0001
0000
11
4º nivel
00001
00000
000000
10
6º nivel
9
8
7
6
5
7º nivel
4
3
2
1
0
9º nivel
10º nivel

FIG. 4

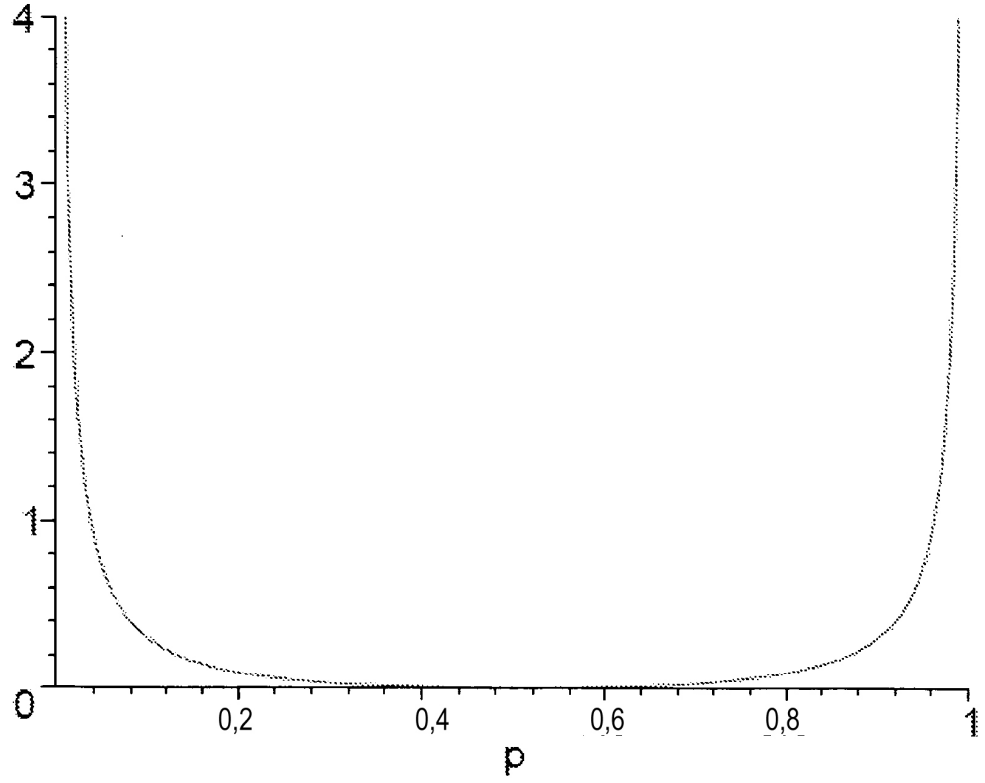

FIG. 5

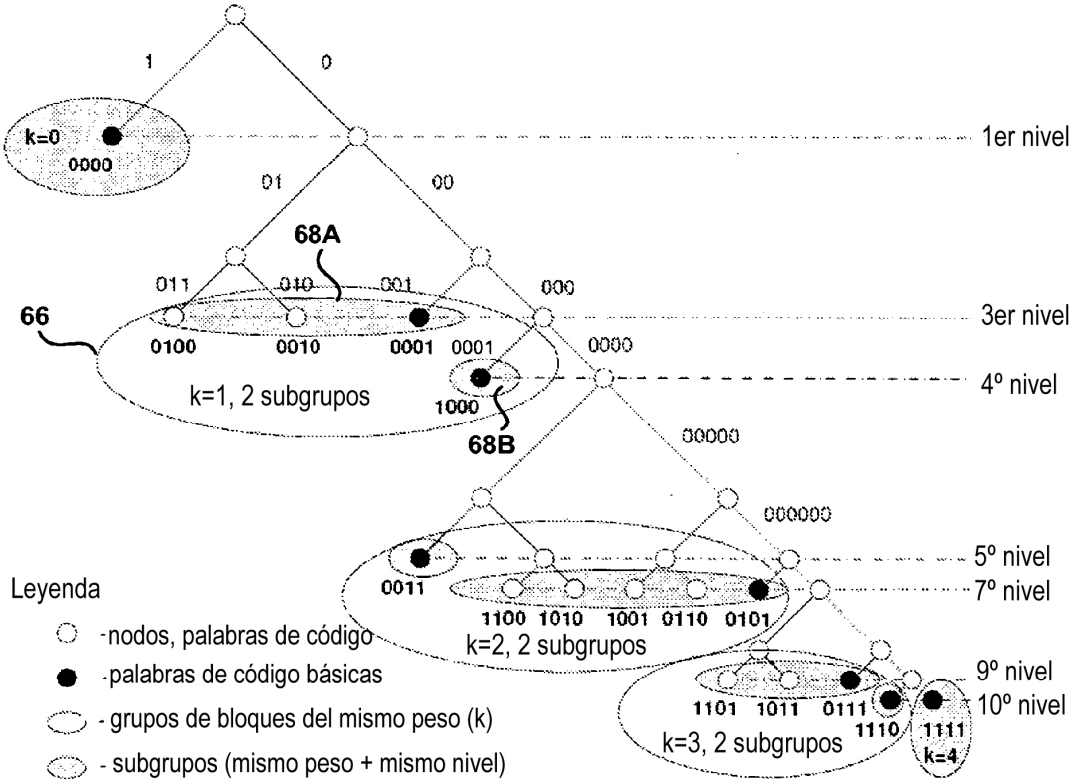
1
0
k=0
0000
1er nivel
01
00
68A
011
010
001
000
66
0100
0010
0001
0001
0000
3er nivel
4º nivel
k=1, 2 subgrupos
1000
68B
00000
000000
5º nivel
7º nivel
0011
1100 1010 1001 0110 0101
k=2, 2 subgrupos
9º nivel
10º nivel
1101 1011 0111
1110
1111
k=4
k=3, 2 subgrupos
Leyenda
-nodos, palabras de código
-palabras de código básicas
- grupos de bloques del mismo peso (k)
- subgrupos (mismo peso + mismo nivel)

FIG. 6

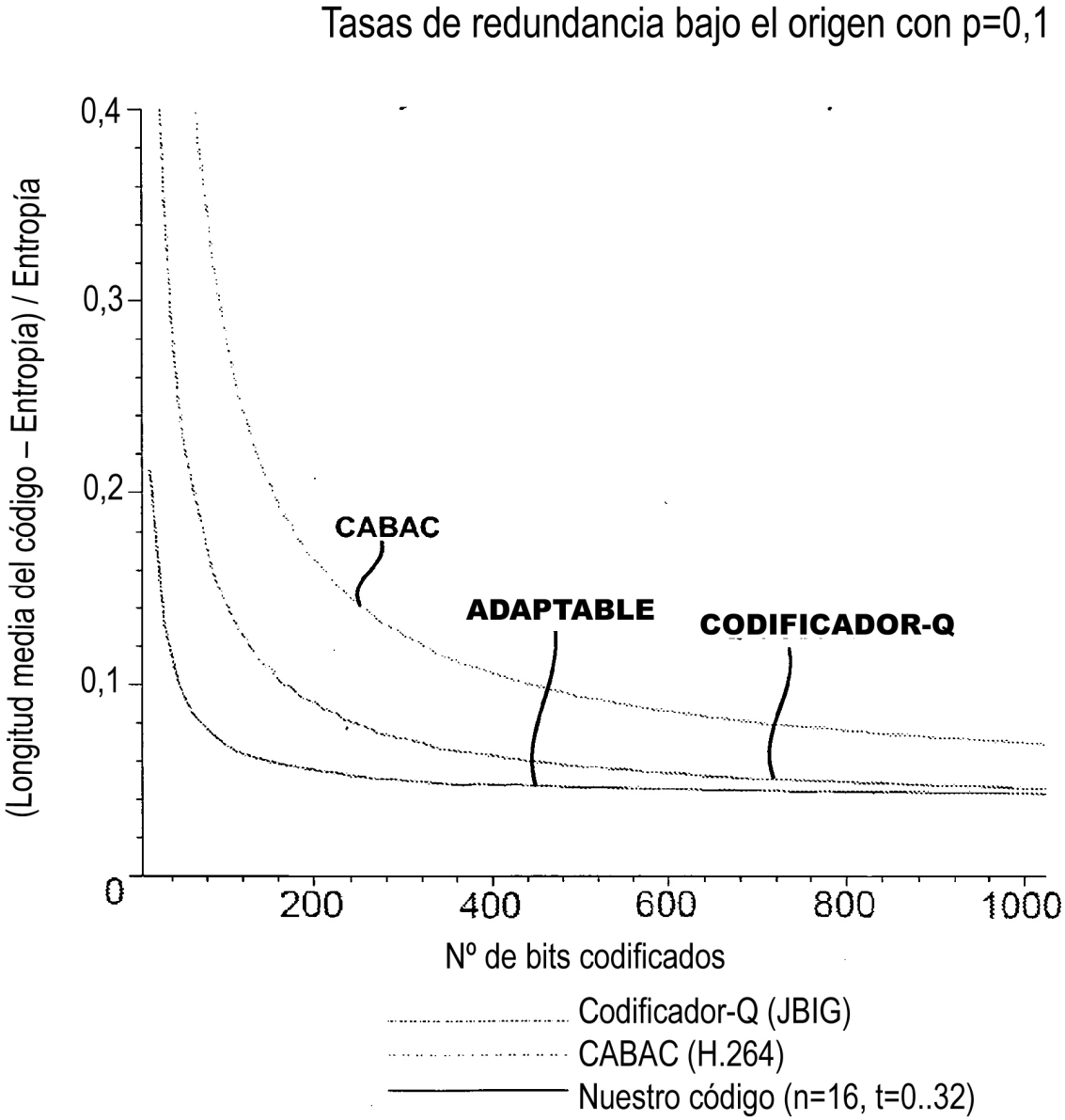
Tasas de redundancia bajo el origen con p=0,1
(Longitud media del código – Entropía) / Entropía
0,4
0,3
0,2
0,1
0
CABAC
ADAPTABLE
CODIFICADOR-Q
200
400
600
800
1000
Nº de bits codificados
Codificador-Q (JBIG)
CABAC (H.264)
Nuestro código (n=16, t=0..32)

FIG. 7A

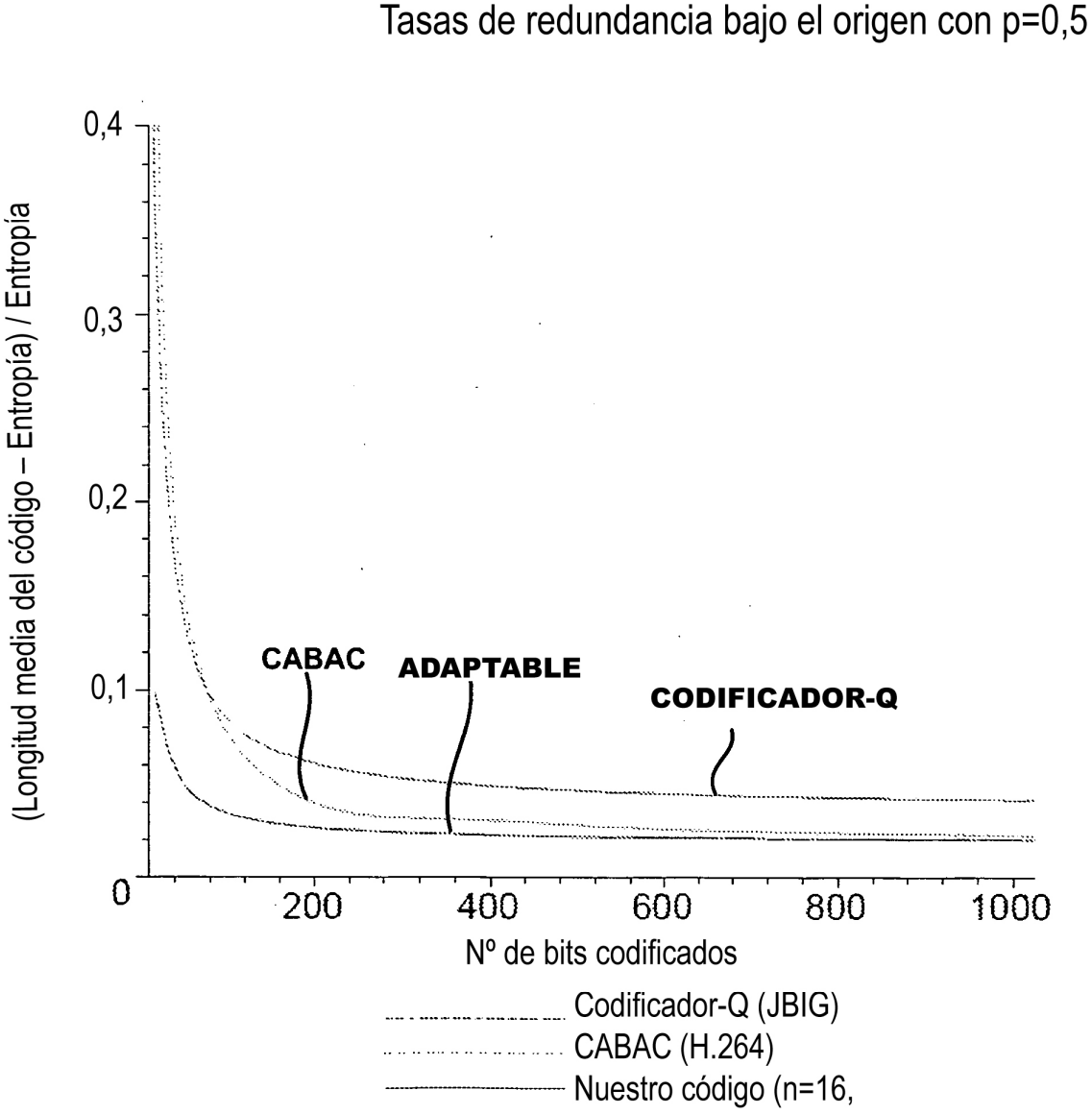
Tasas de redundancia bajo el origen con p=0,5
(Longitud media del código – Entropía) / Entropía
0,4
0,3
0,2
0,1
0
CABAC
ADAPTABLE
CODIFICADOR-Q
200
400
600
800
1000
Nº de bits codificados
Codificador-Q (JBIG)
CABAC (H.264)
Nuestro código (n=16,

FIG. 7B

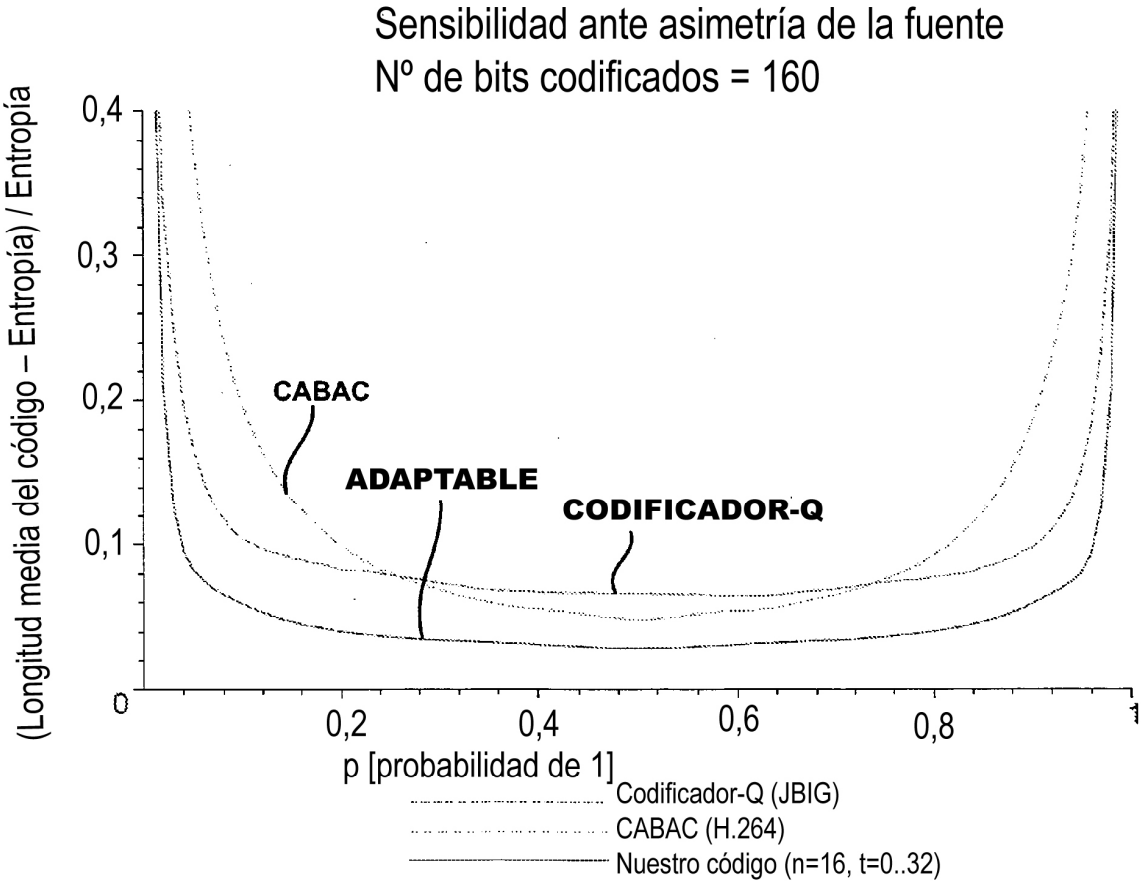
Sensibilidad ante asimetría de la fuente
Nº de bits codificados = 160
(Longitud media del código – Entropía) / Entropía
0,4
0,3
0,2
0,1
0
CABAC
ADAPTABLE
CODIFICADOR-Q
0,2
0,4
0,6
0,8
1
p [probabilidad de 1]
Codificador-Q (JBIG)
CABAC (H.264)
Nuestro código (n=16, t=0..32)

FIG. 8

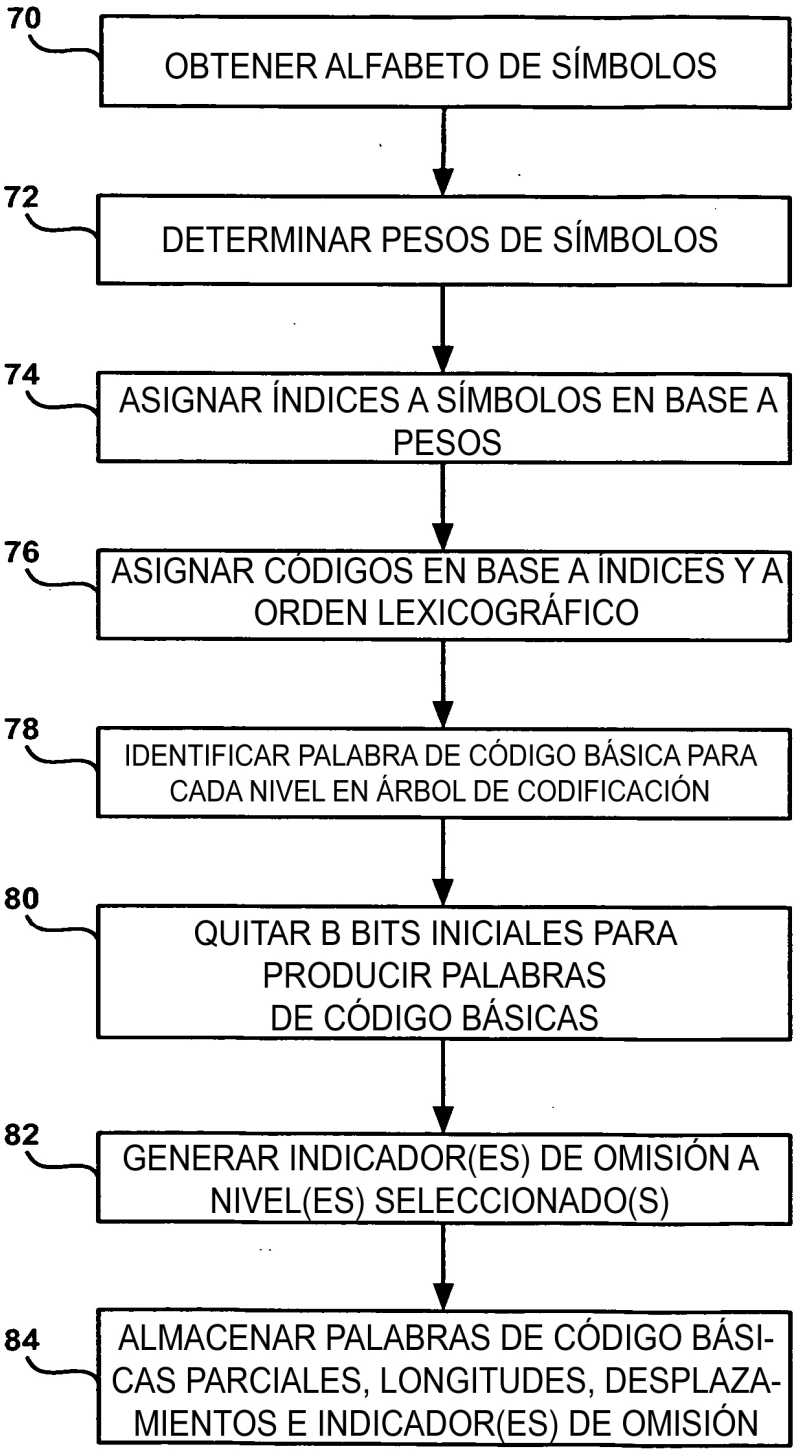
70
OBTENER ALFABETO DE SÍMBOLOS
72
DETERMINAR PESOS DE SÍMBOLOS
74
ASIGNAR ÍNDICES A SÍMBOLOS EN BASE A PESOS
76
ASIGNAR CÓDIGOS EN BASE A ÍNDICES Y A ORDEN LEXICOGRÁFICO
78
IDENTIFICAR PALABRA DE CÓDIGO BÁSICA PARA CADA NIVEL EN ÁRBOL DE CODIFICACIÓN
80
QUITAR B BITS INICIALES PARA PRODUCIR PALABRAS DE CÓDIGO BÁSICAS
82
GENERAR INDICADOR(ES) DE OMISIÓN A NIVEL(ES) SELECCIONADO(S)
84
ALMACENAR PALABRAS DE CÓDIGO BÁSICAS PARCIALES, LONGITUDES, DESPLAZAMIENTOS E INDICADOR(ES) DE OMISIÓN

FIG. 9

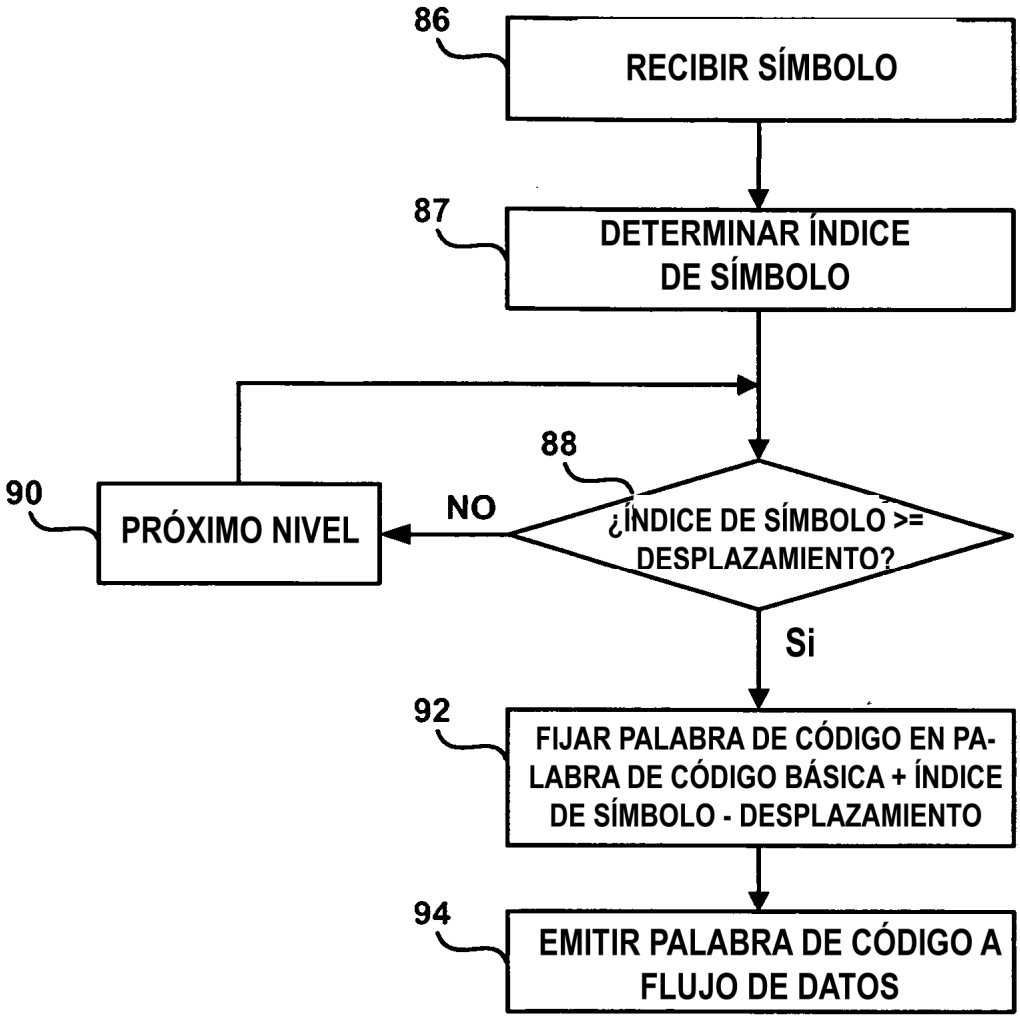
86
RECIBIR SÍMBOLO
87
DETERMINAR ÍNDICE DE SÍMBOLO
88
¿ÍNDICE DE SÍMBOLO >= DESPLAZAMIENTO?
NO
90
PRÓXIMO NIVEL
Si
92
FIJAR PALABRA DE CÓDIGO EN PALABRA DE CÓDIGO BÁSICA + ÍNDICE DE SÍMBOLO - DESPLAZAMIENTO
94
EMITIR PALABRA DE CÓDIGO A FLUJO DE DATOS

FIG. 10

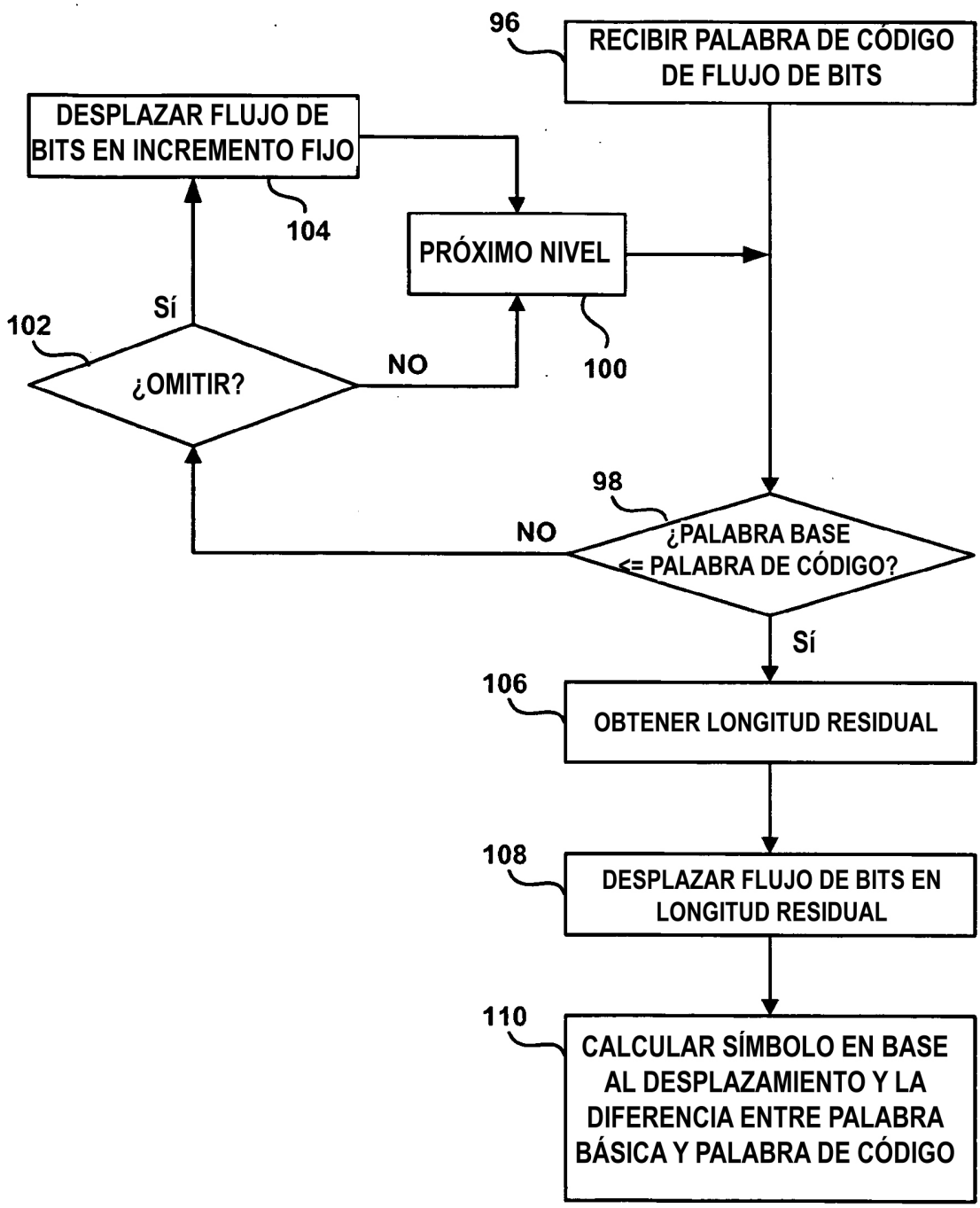
96
RECIBIR PALABRA DE CÓDIGO DE FLUJO DE BITS
DESPLAZAR FLUJO DE BITS EN INCREMENTO FIJO
104
PRÓXIMO NIVEL
100
Sí
102
¿OMITIR?
NO
98
NO
¿PALABRA BASE <= PALABRA DE CÓDIGO?
Sí
106
OBTENER LONGITUD RESIDUAL
108
DESPLAZAR FLUJO DE BITS EN LONGITUD RESIDUAL
110
CALCULAR SÍMBOLO EN BASE AL DESPLAZAMIENTO Y LA DIFERENCIA ENTRE PALABRA BÁSICA Y PALABRA DE CÓDIGO

FIG. 11

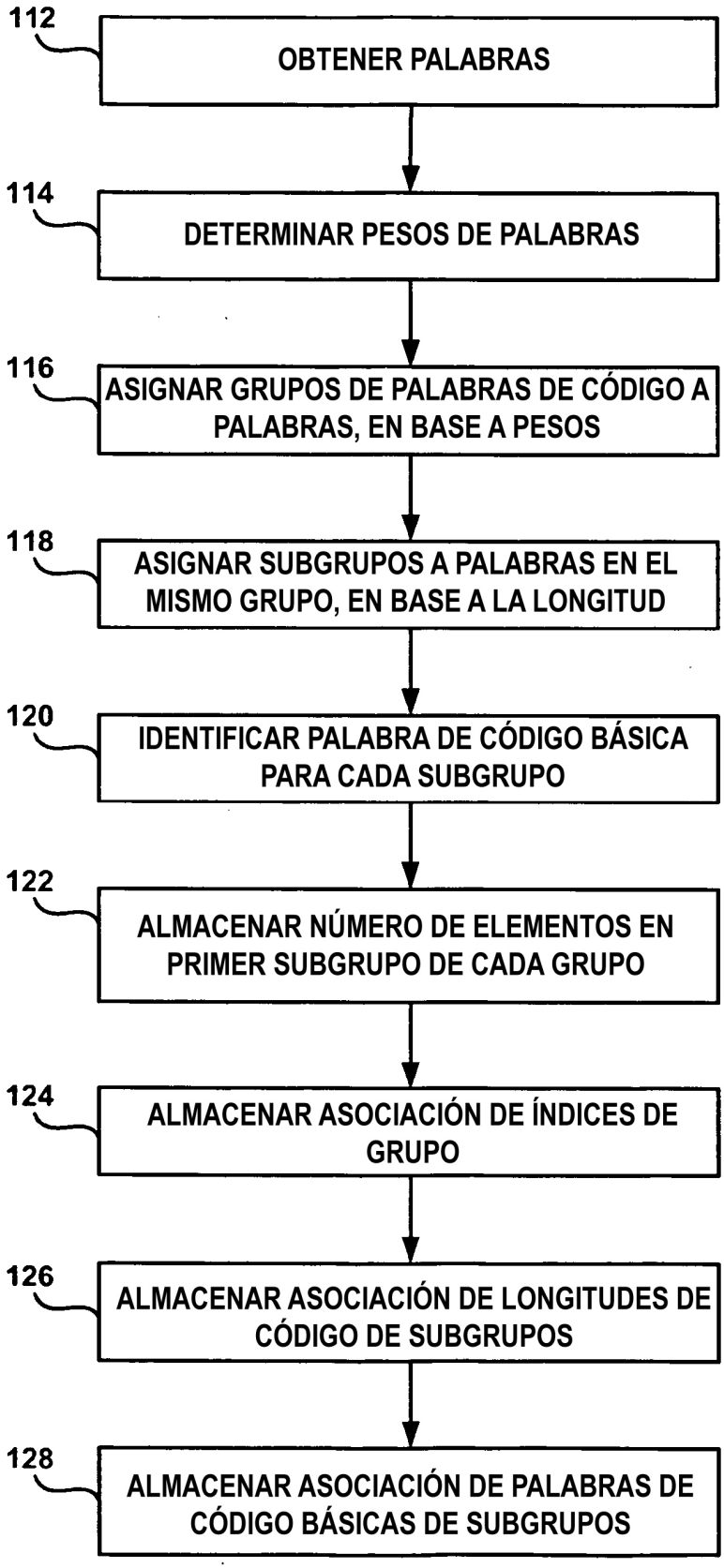
112
OBTENER PALABRAS
114
DETERMINAR PESOS DE PALABRAS
116
ASIGNAR GRUPOS DE PALABRAS DE CÓDIGO A PALABRAS, EN BASE A PESOS
118
ASIGNAR SUBGRUPOS A PALABRAS EN EL MISMO GRUPO, EN BASE A LA LONGITUD
120
IDENTIFICAR PALABRA DE CÓDIGO BÁSICA PARA CADA SUBGRUPO
122
ALMACENAR NÚMERO DE ELEMENTOS EN PRIMER SUBGRUPO DE CADA GRUPO
124
ALMACENAR ASOCIACIÓN DE ÍNDICES DE GRUPO
126
ALMACENAR ASOCIACIÓN DE LONGITUDES DE CÓDIGO DE SUBGRUPOS
128
ALMACENAR ASOCIACIÓN DE PALABRAS DE CÓDIGO BÁSICAS DE SUBGRUPOS

FIG. 12

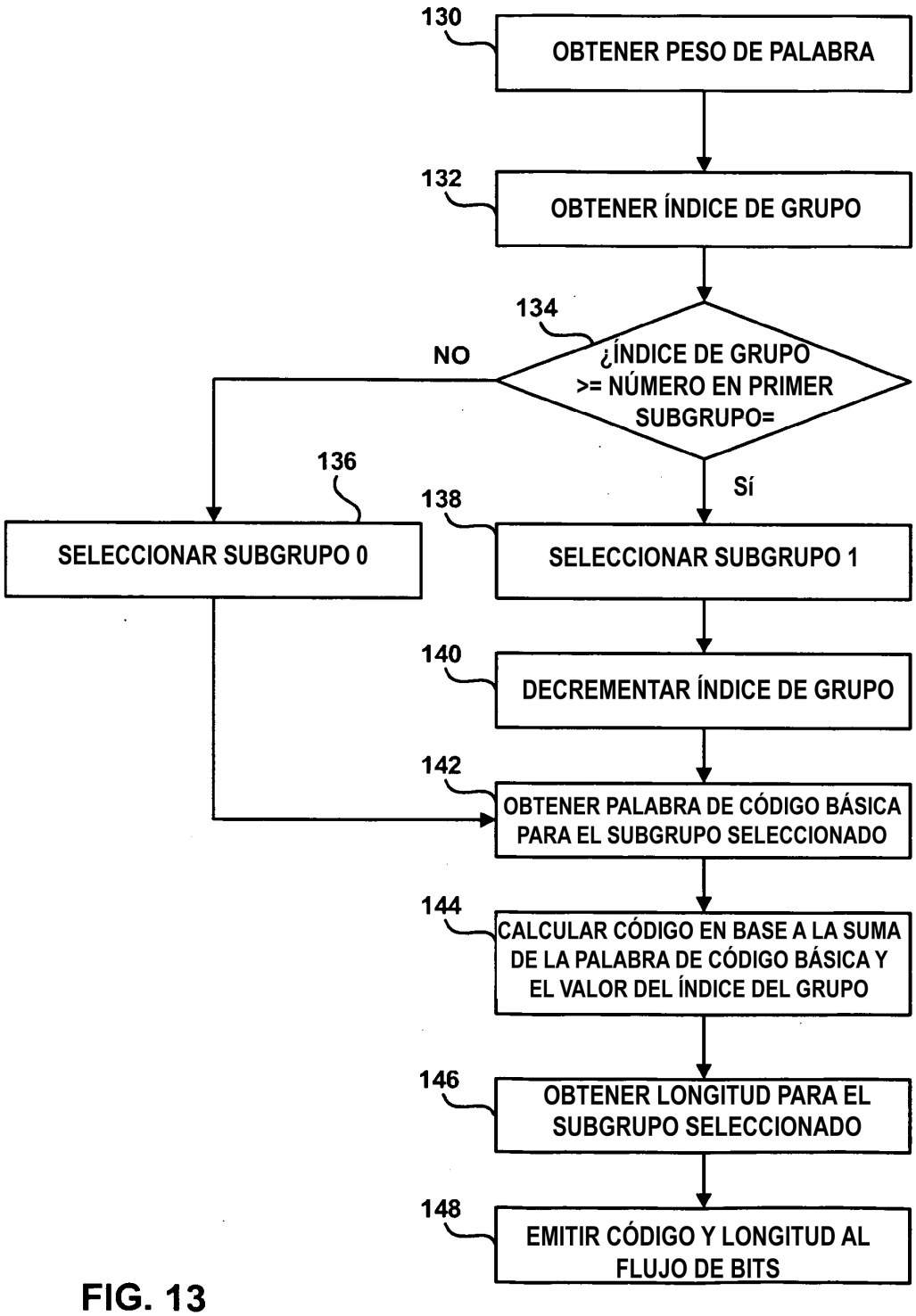
130
OBTENER PESO DE PALABRA
132
OBTENER ÍNDICE DE GRUPO
134
¿ÍNDICE DE GRUPO >= NÚMERO EN PRIMER SUBGRUPO=
NO
Sí
136
SELECCIONAR SUBGRUPO 0
138
SELECCIONAR SUBGRUPO 1
140
DECREMENTAR ÍNDICE DE GRUPO
142
OBTENER PALABRA DE CÓDIGO BÁSICA PARA EL SUBGRUPO SELECCIONADO
144
CALCULAR CÓDIGO EN BASE A LA SUMA DE LA PALABRA DE CÓDIGO BÁSICA Y EL VALOR DEL ÍNDICE DEL GRUPO
146
OBTENER LONGITUD PARA EL SUBGRUPO SELECCIONADO
148
EMITIR CÓDIGO Y LONGITUD AL FLUJO DE BITS

FIG. 13

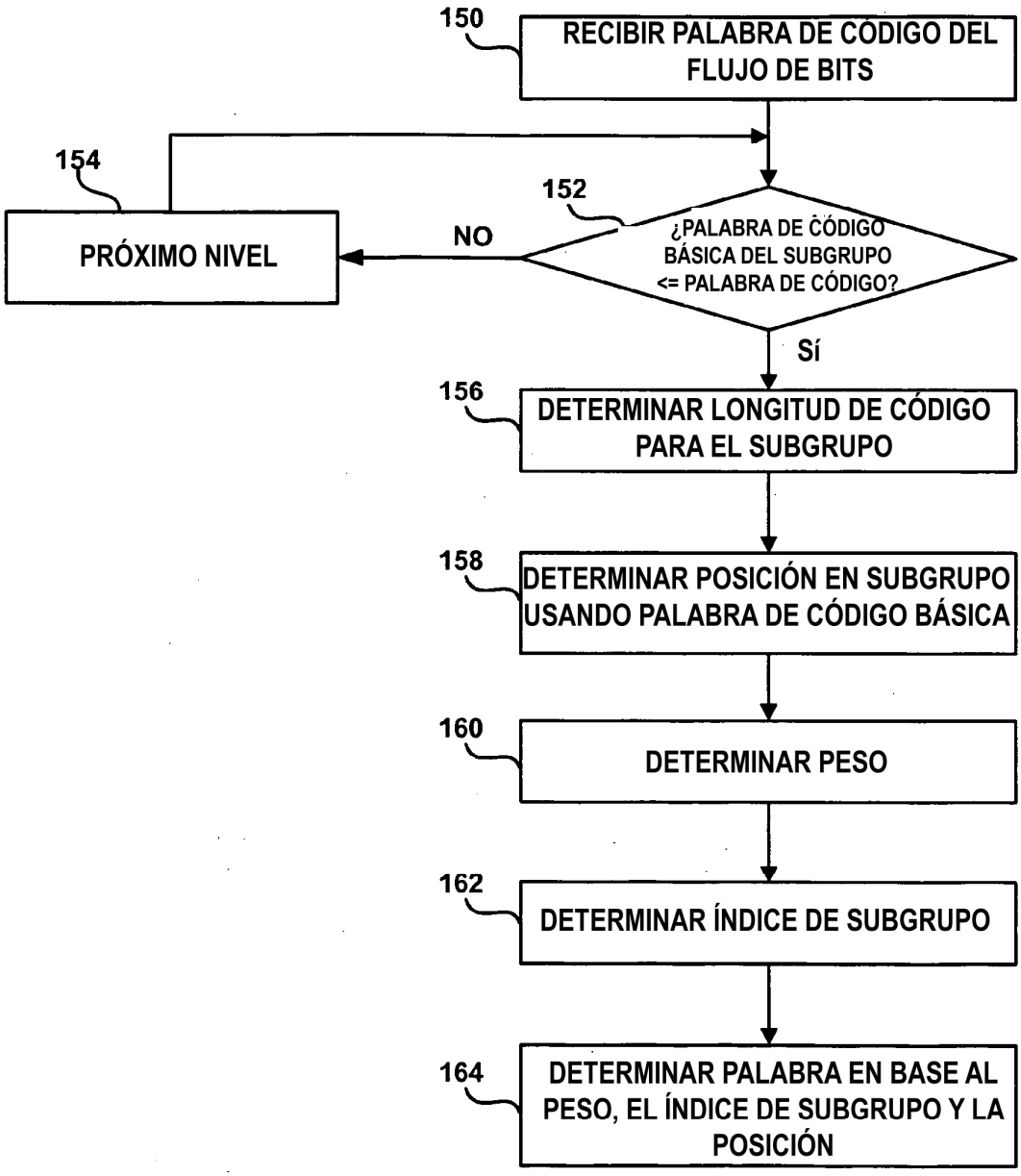
150
RECIBIR PALABRA DE CÓDIGO DEL FLUJO DE BITS
154
152
PRÓXIMO NIVEL
NO
¿PALABRA DE CÓDIGO BÁSICA DEL SUBGRUPO <= PALABRA DE CÓDIGO?
Sí
156
DETERMINAR LONGITUD DE CÓDIGO PARA EL SUBGRUPO
158
DETERMINAR POSICIÓN EN SUBGRUPO USANDO PALABRA DE CÓDIGO BÁSICA
160
DETERMINAR PESO
162
DETERMINAR ÍNDICE DE SUBGRUPO
164
DETERMINAR PALABRA EN BASE AL PESO, EL ÍNDICE DE SUBGRUPO Y LA POSICIÓN

FIG. 14