28
2 Ejem

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

FORMATOS PARA ALMACENAMIENTO Y CODIFICACION DE IMAGENES GRAFICAS

T E S I S

QUE PARA OBTENER EL TITULO DE

A C T U A R I O

P R E S E N T A :

PAULO MAXIMO GUTIERREZ GONZALEZ

ASESOR:
ANA LUISA SOLIS

MEXICO, D. F. ENERO DE 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INDICE GENERAL

PARTE I TECNICAS DE COMPRESION

PARTE II ESTANDARES PARA ALMACENAMIENTO DE IMAGENES

PARTE

PRIMERA

TECNICAS DE COMPRESION

INTRODUCCION

El uso de imágenes con un alto contenido de información es un favorecedor común en áreas de aplicación tales como la arquitectura, multimedia, ingeniería mecánica, diseño industrial, arte y entretenimiento.

En tanto que las aplicaciones basadas en sistemas personales o workstations comunican su información principalmente vía las pantallas de las computadoras, la computación ha jugado un papel central en éste fenómeno, ya que ha llegado a ser una herramienta para la modificación, transmisión y análisis de imágenes muestreadas o generadas por computadora. El scanner está también contribuyendo a incrementar el número de voluminosos archivos gráficos que están saturando los discos de los usuarios y haciendo más lenta la transmisión de imágenes. Desafortunadamente aún no existe un estándar unificado para el almacenamiento y transmisión del gran número de imágenes que están siendo creadas sobre esas plataformas. Como resultado, cada fabricante, desarrollador de software, u organización de servicios de computación ha adoptado un formato de almacenamiento interno de su propiedad [Carl91, Graef89].

Cada una de esas implementaciones considera las partes importantes de la imagen, así como propiedades externas tales como las características de despliegue, y dedica una parte de la estructura de datos a esa información. Muchas de dichas implementaciones también incorporan alguna clase de algoritmo de compresión para minimizar la cantidad de almacenamiento que ese conjunto de datos, inherentemente voluminoso, requiere.

Ha habido contribuciónes significativas a la literatura de compresión de imágenes y técnicas de cuantificación para reducir el ancho de banda en la transmisión de datos [CDF86, GW87, Hec82, LS71, Stor92, Win72, WP71]. Esa clase de técnicas usualmente cambian la imagen en algún umbral de aceptabilidad para codificar los datos. Existen también algoritmos que pueden comprimir la imagen en una forma tal que al descomprimirla la reproducen exactamente como existió antes del proceso de compresión.

La primera parte de éste trabajo de tesis revisa las diferentes clases de algoritmos de compresión. En la segunda parte se presentan algunos de los formatos de almacenamiento más comúnmente utilizados. Esto no significa una cobertura exhaustiva de tales formatos, ni se intenta hacer ningún juicio acerca de cuál enfoque es más o menos apropiado. Por último en la parte tres se diseñan modulos de captura y despliegue de imágenes RGB en los formatos IMG, PCX y GIF.

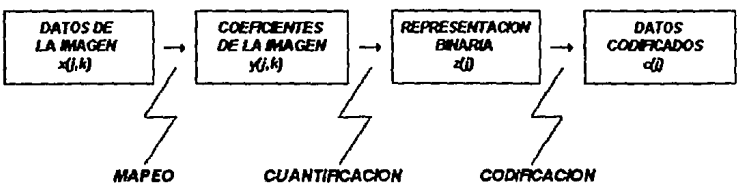
EL PROCESO DE COMPRESION DE LA IMAGEN

Las técnicas de compresión de datos, en general, están divididas en dos clases: Las no recuperables y las recuperables.

Las técnicas no recuperables usualmente conceden una cierta pérdida de información (en un umbral de aceptabilidad) mediante un proceso de cuantificación de los datos, para ganar una mayor tasa de compresión; incluso existen técnicas de ésta clase que pueden ser ajustadas a diferentes niveles de calidad (umbrales de aceptabilidad) ganando así, mayor calidad en detrimento de la tasa de compresión. Esta clase de técnicas son efectivas al aplicarse a imágenes gráficas y sonidos digitalizados, ya que la naturaleza análoga de éstos permite que sea más aceptable la idea de que la entrada no es del todo igual a la salida.

Por otra parte, las técnicas recuperables, son aquellas que nos garantizan que después del proceso de descompresión de los datos, estos resultarán como la entrada original, es decir, reproducen la entrada original tal y como existió antes de comprimirla. Estas técnicas son apropiadas para datos tales como archivos generados en hojas electrónicas de cálculo, manejadores de bases de datos y procesadores de palabras, ya que en ésta clase de información el perder un bit podría ser catastrófico. El proceso de decodificación en este tipo de técnicas, puede ser considerado lógicamente como el inverso del proceso de codificación, aún cuando en una implementación se involucren pasos y operaciónes adicionales.

La compresión/descompresión de imágenes gráficas puede considerarse como un proceso de codificación/decodificación de los bytes de datos que representan a la imagen . En la siguiente figura se muestra un diagrama funcional del proceso de codificación de una imagen [GW87]:



Usualmente, las técnicas recuperables no consideran la etapa de cuantificación, puesto que en general no es reversible

MAPEO

La etapa de *mapeo* convierte la información espacial en información espectral o de frecuencia, es decir, convierte la información de la imagen, la cual viene dada por el valor de un pixel en un punto (que en general es un valor de 8 bits para

cada plano de color) de la pantalla de dimensiones X y Y, a información en el dominio de la frecuencia (lo que se lleva a cabo aplicando una función de procesamiento de señales de alto nivel), después de lo cual se pueden identificar las piezas de información menos relevantes (lo que es difícil de hacer a partir de la imagen espacial) y; debido a que muchas imágenes gráficas sobre las pantallas de las computadoras están compuestas de información de baja frecuencia, esto es muy significativo, ya que mediante esta etapa podemos identificar las piezas de información que podemos desechar sin comprometer seriamente la calidad de la imagen.

CUANTIFICACION

Una vez identificadas las piezas de información que pueden ser desechadas, se procede a hacerlo, utilizando alguna técnica de ***cuantificación***, que no es otra cosa que reducir el número de bits que emplea un número entero, reduciendo así su precisión. Desde luego, al ser ésta la parte no recuperable del proceso de codificación de la imagen (por cuanto lleva compactación adicional), se deben establecer niveles de aceptabilidad, los cuales generalmente están basados en estudios estadísticos y en algunas implementaciónes, se da la oportunidad al usuario de decidir dicho nivel al tiempo de corrida.

Es importante recordar que por ser ésta una etapa no reversible, las técnicas recuperables no la consideran.

CODIFICACION RECUPERABLE

Finalmente, los datos (cuantificados en el caso de las técnicas no recuperables o los de entrada en el caso de las técnicas recuperables) son comprimidos utilizando alguna técnica recuperable convencional.

Este tipo de técnicas éstan clasificadas de acuerdo al modelo utilizado para codificar los datos, es decir, el proceso de ***codificación*** consiste en la aplicación de un codificador basado en un modelo, el cual es un conjunto de reglas y datos utilizados para procesar los símbolos de entrada y decidir el código de salida.

MODELADO

Para esclarecer los conceptos anteriores considerar el algoritmo de Huffman [Huf52], el cual basado en probabilidades de ocurrencias preestablecidas (modelado estadístico) para cada dato en el flujo de entrada, genera un árbol binario, siendo las hojas de este todas las posibles probabilidades para todos los posibles datos de entrada.

Entonces, el código para un dato de entrada cualquiera, se obtiene recorriendo el árbol de la raíz a la hoja de probabilidad asociada a dicho dato (proceso codificador).

De lo anterior debiera intuirse que si se cambian las probabilidades de ocurrencia para los datos de entrada (es decir, si se cambia el modelo), también cambiarán los códigos de salida, siendo el codificador el mismo que antes (el proceso de recorrer el árbol de la raiz a una hoja para obtener el código de un dato cualquiera).

Algo que es evidente, es que entre mejor se ajusten los datos de entrada a las probabilidades dadas, mejor será la codificación. Por el contrario, si las probabilidades no reflejan la naturaleza de los datos, la salida podría incluso (en un caso extremo) resultar en códigos más largos que los de entrada, lo cual, claramente va en contra de los objetivos trazados.

De lo anterior se puede ver que el modelado juega un papel importante en la codificación. De hecho, el modelado es lo que la máquina a un automovil y el codificador vendrían siendo las ruedas.

Regularmente las técnicas recuperables convencionales se implementan utilizando modelado estadistico o esquemas de diccionario:

MODELADO ESTADISTICO

Un modelo estadistico lee y codifica un dato de entrada a la vez, utilizando la probabilidad de ocurrencia de dicho dato.

Se ha mencionado ya la forma más simple del modelado estadistico, la que está basada en una tabla fija de probabilidades. También se hizo la observación de que dicha tabla debería ajustarse lo suficiente a la naturaleza de los datos a comprimir para obtener una compresión efectiva.

En el pasado, debido a impedimentos de procesamiento de la CPU, se utilizó una tabla fija de probabilidades universal, es decir, se utilizó la misma tabla de probabilidades para cada flujo de entrada. Aunque dicha tabla fué calculada a partir de flujos de datos "representativos", es evidente que al hacerlo de esta forma se agudizan los problemas de ajuste entre ésta y la naturaleza de los datos de entrada, pues no es lo mismo utilizar una tabla universal para un flujo dado, que una tabla que refleje las estadisticas de dicho flujo, por lo que el paso siguiente fué implementar una tabla de estadisticas para cada flujo de entrada.

En este punto, el modelado estadistico determina la probabilidad de ocurrencia del dato que actualmente ingresa, a partir de los datos que ingresaron previamente. El número de datos previos a utilizar para calcular dicha probabilidad está determinado por el orden del modelo, es decir, un modelo de orden cero, no toma en cuenta los datos previamente ingresados, un modelo de orden uno, toma en cuenta solo el último dato que ingreso,etc.

Debido a esta forma de calcular las probabilidades, éstas pueden variar mucho de un modelo a otro; por ejemplo, si en un modelo de orden cero ingresa una 'u' su probabilidad sería del uno por ciento (asumiendo 100 caracteres

equiprobables); y si en este mismo flujo implementamos un modelo de orden uno y antes de la 'u' apareció una 'p', la probabilidad para la 'u' sería de noventa y cinco por ciento (1 - (ord('u') - ord('p')) / 100 = 1 - 5 / 100 = .95).

Claramente, entre mayor información previa sea tomada (es decir, entre mayor sea el orden del modelo), mejor será el modelo.

Desgraciadamente, el incrementar el orden del modelo hace que las estadísticas, las cuales evidentemente necesitan ser almacenadas junto a los datos codificados porque serán utilizadas por el decodificador, crezcan en una tasa muy grande, lo que muy probablemente obscurecerá cualquier ganancia en mejorar la tasa de compresión.

Por ésta razón, en la última década la investigación se ha concentrado en modelos adaptativos, los cuales no necesitan revisar los datos previamente para generar las estadísticas, sino que las estadísticas son continuamente modificadas cada vez que son leídos y codificados nuevos datos.

El problema de los modelos adaptativos es que al iniciar la codificación no se sabe prácticamente nada acerca de la naturaleza de los datos, por lo que al inicio no hacen un buen trabajo de compresión. Aunque existen algoritmos adaptativos que tienden a ajustarse rápidamente a los datos, obteniendo tasas de compresión respetables después de sólo unos cientos de bytes.

ESQUEMAS DE DICCIONARIO

Cuando uno lee un artículo, muy a menudo se encuentra con cosas como:

"El formato estándar TIFF [Tag] fue diseñado principalmente para...".

Y se sabe que [Tag] es una referencia de la cual se encontrará información detallada al final del artículo, en una lista (diccionario) de referencias. El hacerlo de ésta forma evita tener que escribir toda la información de la referencia cada vez que sea necesario citarla.

Los esquemas de diccionario utilizan este concepto para codificar los datos de entrada:

Cada vez es leída una cadena de datos en el flujo de entrada y a continuación, ésta es buscada en una tabla estática o diccionario de cadenas. Si es encontrada, el código para dicha cadena será su índice o apuntador en el diccionario. Claramente, entre mayor sea la longitud de la cadena, mayor será la tasa de compresión.

Cabe señalar que por su naturaleza, los esquemas de diccionario le dan mucha más importancia al modelado que a la codificación, la cual simplemente consiste en producir el índice o apuntador (de longitud fija) como el código para una cadena.

Nuevamente, el problema con una tabla estática es que ésta necesita ser almacenada junto con los datos codificados (de la misma forma que la lista de referencias es adjuntada al artículo que la utiliza).

Los esquemas de diccionario adaptativos evitan este problema al generar el diccionario al tiempo de llevar a cabo la codificación. Un ejemplo de un esquema de diccionario adaptativo se da al utilizar acrónimos:

"El joint en JPEG se refiere a la colaboración entre la CCITT y la ISO".

En ésta frase solo bastó que la primera vez que se citaron los acrónimos JPEG (Joint Photographic Experts Group), CCITT (International Consultive Committee on Telegraph and Telephone) e ISO (International Standars Organization), se mencionara también su significado (como se está haciendo aquí) para que la próxima vez que fueran encontrados se reprodujera mentalmente su significado:

"Esta tesis hace un estudio del estándar JPEG propuesto para comprimir imágenes".

CONTENIDO DE INFORMACION

El objetivo principal de la compresión de datos, en general, es eliminar la información redundante en un mensaje dado. Entonces, para comprimir un mensaje, es un buen inicio tener una medida de la cantidad de información proporcionada por este. Por lo cual se ha adoptado de la termodinámica la palabra entropía para denotar dicha cantidad (ya que en la termodinámica este concepto tiene un significado similar), es decir, la entropía de un mensaje es la cantidad de información proporcionada por este, por lo que, entre mayor sea la entropía de un mensaje, mayor será la información que nos proporcione.

A un nivel atómico, la entropía de un dato (el cual puede ser una letra en un texto o un pixel en una imagen) está dada por el negativo del logaritmo de la probabilidad de ocurrencia de dicho dato y; la entropía de un mensaje entero es la suma de éstas entropías atómicas. Si la información está dada en bits el logaritmo utilizado es el logaritmo en base dos.

Por ejemplo: Supongase que en un texto la probabilidad de ocurrencia de la letra 'a' es de 1/32. Entonces, entropía('a') = -log2(p('a')) = -log2(1/32) = (-ln(1/32)) / ln(2) = 5. Esto significa que para codificar la 'a' necesitamos de solo 5 bits, en lugar del estandard ASCII de 8 bits.

Al haberse acuñado en el pasado el concepto de entropía, el siguiente paso fué codificar el mensaje original con la cantidad de bits demandados por la entropía de este . Fue así como se concibieron los algoritmos de codificación de Shannon-Fano y de Huffman, este último ya mencionado con anterioridad.

La codificación de Huffman conduce a la mínima cantidad de redundancia posible al utilizar códigos de longitud variable [Huf52]. Debido a que la longitud de dichos códigos es un número entero, la codificación de Huffman no es óptima, pero sí la mejor aproximación a la entropía del dato que se está codificando.

Para aclarar lo anterior supongase que en el ejemplo anterior la probabilidad de ocurrencia de la 'a' no es de 1/32, sino de 1/10 (es decir, supongase que se utiliza otro modelo), entonces la entropía de 'a' también cambiará de 5 a 3.32 (-log2(1/10)) y, en este caso, la cantidad de bits necesarios para codificar la 'a' no es de 3.32 sino de 3 o 4 (un número entero).

Posteriormente, en busca de una mayor eficiencia se concibió al sucesor natural del algoritmo de Huffman (y también debido a que bajaron los costos del procesamiento de la CPU): La codificación aritmética.

La codificación aritmética a diferencia de la de Huffman no produce un código para cada dato, sino que produce un código para el mensaje entero: Cada vez que un dato es ingresado el código de salida es modificado incrementando su longitud, pudiendo de ésta forma sumar el efecto neto del dato al código. En el ejemplo anterior, se sumará la entropía exacta de 3.32 de la 'a' al código de salida (en [Mark92] se dan implementaciones tanto de la codificación aritmética como de otros compresores).

La codificación aritmética requiere de mayor poder de procesamiento, con el consiguiente costo adicional, pero es un costo que vale la pena pagar cuando su contraparte de almacenamiento o transmisión es bastante mayor.

ALGORITMOS RECUPERABLES PARA COMPRIMIR IMAGENES

Una imagen generada por computadora queda almacenada en un Refresh Buffer (Buffer de memoria que es desplegado cíclicamente para evitar el parpadeo originado por el decaimiento del fósforo en un dispositivo de salida tal como el CRT) en términos de sus puntos componentes, llamados pixels o pels (picture elements). La imagen está formada por el rastro (raster) dejado en la pantalla al desplegar el contenido del refresh buffer. El raster a su vez está formado por un conjunto de líneas horizontales compuestas de pixels, llamadas líneas raster. El raster es de esta forma una matriz de pixels que cubre la pantalla entera.

Por otra parte, un pixel es almacenado en el refresh buffer como un conjunto de bits que, en general, consta de tres partes (cada una de 8 bits y; en imágenes monocromas, solo una parte, de un bit) cada una describiendo la cantidad de cada color primario (rojo, verde y azul, para un esquema de mezclado de colores RGB) presente en el pixel, por lo que a éste tipo de imágenes también se les llama de mapa de bits (ya que un conjunto de bits está en correspondencia uno a uno con un punto en la pantalla) [Foley].

Las técnicas recuperables convencionales toman como flujo de datos a la imagen entera, siendo un dato, un pixel en la imagen.

En las secciónes siguientes se describirá el algoritmo básico en el cual están basadas algunas de las técnicas recuperables convencionales de mayor uso para comprimir imágenes.

ESQUEMA DE CODIFICACION DE HUFFMAN

En la decada de los 40's al gestarse los conceptos de entropía (contenido de información) y redundancia (diferencia entre el volumen de un mensaje y la información que realmente contiene), los investigadores pensaban que si se conocía la probabilidad de los datos en un mensaje dado, debería ser posible codificar dichos datos de forma tal que el mensaje ocuparía menos espacio.

Si se toma en cuenta que hasta después de la segunda guerra mundial no había aún computadoras digitales, se puede ver que la idea de desarrollar algoritmos para codificación utilizando aritmética binaria fué un gran adelanto.

En esa línea fué como se dió el primer algoritmo para codificación ampliamente conocido: El algoritmo de Shannon-Fano (el nombre es debido a que dicho algoritmo fué desarrollado casi al mismo tiempo en los laboratorios de la compañía de telefonos norteamericana "Bell" por Claude Shannon* y en el Instituto Tecnológico de Massachusetts por R.M. Fano) el cual en un intento por minimizar el número de bits usados en un mensaje, utiliza las probabilidades de los datos en el mensaje para construir una tabla de códigos correspondientes.

La tabla es construida a partir de un árbol binario recorriendo éste de la raíz a cada una de sus hojas y obteniendo de esta forma en cada paso un código.

El algoritmo para construir el árbol es simple:

ALGORITMO DE SHANNON-FANO

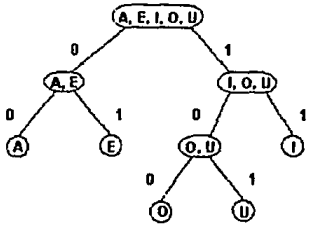
PASO 1 Construir una tabla con las frecuencias relativas (o en un caso ideal, con las probabilidades) de todos los posibles datos en el mensaje de entrada.
PASO 2 Ordenar la lista de datos de acuerdo a sus frecuencias relativas de mayor a menor.
PASO 3 Dividir la lista anterior en dos de modo que la suma de frecuencias relativas en ámbas partes sea lo más parecida posible.
PASO 4 Asignar a la lista superior (la que tiene mayores frecuencias relativas) el 0 y a la inferior el 1.
PASO 5 Repetir los pasos 3 y 4 sobre cada lista hasta que todas las listas posibles consten de un solo elemento.

Para aclarar el funcionamiento del algoritmo anterior supongase que cualquier mensaje solo puede contener las cinco vocales y que de un mensaje dado se ha obtenido la siguiente tabla de frecuencias:

DATO	FRECUENCIA
A	19
E	10
I	8
O	8
U	7

* Conocido como el padre de la teoría de la información por su trabajo dado en los 40's y 50's Shannon definió los conceptos de contenido de información y entropía como relativos a datos

Se ha ya dado el primer y segundo pasos del algoritmo, por tanto si se continúa se obtiene el siguiente árbol:



y la correspondiente tabla de códigos es:

DATO	CODIGO
A	00
E	01
I	11
O	100
U	101

La virtud de este proceso es que a los datos más probables les asigna códigos más cortos (con la consiguiente reducción de bits necesarios para transmitir el mensaje) por lo que este esquema de codificación va de acuerdo a la noción de entropía de un dato. Recordando que la entropía de un dato está dada por el negativo del logaritmo de la probabilidad de ocurrencia del dato, entre más probable sea dicho dato, su entropía será menor (ya que en (0,1) $f(x) = -\log(x)$ es decreciente), es decir, un dato que aparece con mayor frecuencia en un mensaje, es más redundante que otros, por lo que nos comunica menos información, lo cual se refleja al asignarle un código más corto.

Para el ejemplo se tienen los resultados siguientes:

DATO	FREC.	Cont. de inf. en bits $-LOG_2(FREC./52)$	Bits de inf. para el dato en el mensaje dados por su entropía	Cont. de inf. por SF en bits	Bits de inf. para el dato por SF
A	19	1.4525122	27.597732	2	38
E	10	2.3785116	23.785116	2	20
I	8	2.7004397	21.603518	2	16
O	8	2.7004397	21.603518	3	24
U	7	2.8930848	20.251594	3	21
	TD = 52		EM = 114.84148		ESF = 119

en donde. si el mensaje viene en código ASCII ocupa sin compresión TD X 8 = 416 bits. requiriendo la entropía de éste de solo 114.84148 bits y Shannon-Fano de 119 bits.

Lo anterior muestra que en este caso, la codificación de Shannon-Fano es bastante buena ya que da una tasa de compresión efectiva del 71.39 % (1 - 119 / 416), siendo la tasa óptima del 72.39 % (1 - 114.84148 / 416).

Solo de que existan al menos tantos símbolos d de codificación (a los que se les llama el alfabeto de códigos) como datos posibles n en cualquier mensaje (en el caso binario d = 2 y el alfabeto de códigos son los dígitos binarios y; en el caso de archivos de texto n = 256 y los posibles datos son el conjunto ASCII), para un dato, su código correspondiente constará, en general, de más de un símbolo de codificación y; si definimos la longitud de dicho código como el número de símbolos de codificación que utiliza, entonces dicha longitud ha de ser lo más pequeña posible si el código ha de ser óptimo. De hecho, dicha longitud ha de acercarse por arriba cuanto sea posible a la entropía del dato, la cual como ya vimos, depende de la probabilidad asignada a éste. Por lo que un código óptimo, en general, constará de códigos de longitud variable.

Debido a que el algoritmo de Shannon-Fano utiliza un número entero de dígitos binarios para codificar un dato cualquiera no puede, en general, minimizar el número de bits utilizados para transmitir un mensaje, ya que este número podría ser fraccionario.

Supongase que para cualquier mensaje hay n posibles datos y que $p(i)$ es la probabilidad del $i-$ésimo dato con longitud de código $l(i)$ entonces, la longitud promedio del mensaje μl estará dada por la media ponderada de las longitudes $\mu l = \Sigma_i p(i)\ l(i)$ en donde $i \in \{1,2,...,n\}$.

De lo anterior es evidente que un código óptimo o de mínima redundancia es aquel para el cual μl es mínima o equivalentemente un código óptimo es aquel para el cual $l(i)$ es mínima $\forall i \in \{1,2,...,n\}$.

A pesar de que la codificación de Shannon-Fano no es óptima, a medida que el número de datos en el mensaje se aproxima a infinito, ésta se aproxima al comportamiento óptimo. Aunque existe la codificación de Huffman que en general es mejor que la de Shannon-Fano.

Cuando David A. Huffman publicó su artículo en septiembre de 1952 [Huf52], tal vez no imaginó que sería (y probablemente lo sigue siendo) el artículo más citado y que ha causado una cantidad importante de investigaciónes en la teoría de la información.

La manera en que Huffman dedujo su esquema de codificación es un tanto natural. El partió del hecho de que si un código es óptimo (en el sentido de mínima redundancia) entonces cumple una serie de condiciones, las más obvias de las cuales son:

I) Dos datos cualesquiera no tendrán el mismo código.

II) Los códigos serán tales que no serán necesarias indicaciones adicionales para señalar dónde empieza y dónde finaliza un código una vez que el punto de partida de una secuencia de códigos es conocido.

Si es definido el prefijo de orden k de un código de un dato como los primeros k dígitos de ese código entonces la condición II implica la condición llamada de "código prefijo", la cual establece que un código para un dato cualquiera no será prefijo de ningun orden de otro código para otro dato, de lo contrario dada una secuencia de códigos habría que, darle información adicional al decodificador acerca de éstos (tal como sus longitudes), ya que si por ejemplo se tienen los códigos 11,111,102 y 02, el decodificador no sabría al recibir una secuencia tal como 11102, si se trata del mensaje 111-02 o del mensaje 11-102, si solo se le indica el inicio de una secuencia tal.

Es importante en el ejemplo anterior resaltar que aunque 02 también es subcadena de otro código (el 102) no es subcadena prefijo (con esta terminología se diría que es subcadena sufijo) por lo que no representa un problema serio para el decodificador ya que, asumiendo la condición de código prefijo, un 1 antes del código 02 no puede por sí solo ser un código (porque es prefijo de 102) por lo que necesariamente es la terminación de un código más largo, por ejemplo 111 u 11. Lo que es más, la existencia como código de alguno de los 2 anteriores automaticamente elimina al otro, ya que 11 es el prefijo de orden 2 de 111.

Se ha visto ya que la longitud promedio μ_l de un mensaje cualquiera está dada por $\mu_l = \sum_i p(i)\ l(i)$ en donde $i \in \{1,2,...,n\}$. Entonces:

Dados $i \neq j$ con $p(i) \geq p(j)$ (y como $1 \geq p(i) \geq 0$ y $l(i) \geq 0\ \forall\ i \in \{1,2,...,n\}$) implica que, si el código ha de ser óptimo $l(i) \leq l(j)$.

Es decir, la longitud del código de un dato i no puede ser mayor que la longitud del código de otro dato j si i es más probable que j, ya que si ésto sucediera, sería posible reducir μ_l intercambiando los códigos entre i y j:

$$p(i) \geq p(j)\ y\ l(i) \geq l(j) \Rightarrow p(i)\ l(j) + p(j)\ l(i) \leq p(i)\ l(i) + p(j)\ l(j).$$

Además, si $p(i) = p(j)$ entonces $p(i)\ l(j) + p(j)\ l(i) = p(i)\ l(i) + p(j)\ l(j)$, es decir, dos datos con la misma probabilidad pueden diferir en longitud y eso no afecta la longitud promedio del mensaje μ.

De lo anterior se concluye que si un código es óptimo, se cumple que:

III) $p(1) \geq p(2) \geq \ldots \geq p(n) \Rightarrow l(1) \leq l(2) \leq \ldots \leq l(n)$.

Dados los datos i e $i + 1$ con $p(i) \geq p(i + 1)$ como se ha visto para un código óptimo ésto implica $l(i) \leq l(i + 1)$. Supongase ahora que en la última relación se da la desigualdad estricta: ésto significa que se la han asignado q ($q > 0$, $q \in Z$) dígitos más del alfabeto de códigos al dato $i + 1$ y, como el código es óptimo, el prefijo de orden $l(i)$ del código del dato $i + 1$ no es utilizado como código para ningun otro dato. Entonces, uno podría razonar diciendo: si lo anterior sucede, ¿porque no asignar el prefijo de orden $l(i)$ del código del dato $i + 1$ como el código de dicho dato?, es decir, ¿porque no eliminar los q dígitos últimos del código del dato $i + 1$? ya que ésto reduciría μ. La respuesta es que si bien dicho prefijo no es utilizado como código para otro dato, no existe impedimento para que sea prefijo de otro código distinto del código para el dato $i + 1$ (tal vez de mayor longitud). Aunque si $i = n - 1$, el prefijo de orden $l(i) = l(n - 1)$ del dato $i + 1$ solo lo es del dato $i + 1$, ya que no existe otro código de longitud mayor o igual que $l(i + 1) = l(n)$. Por lo que para este caso, si es posible eliminar los q dígitos últimos del código del dato $i + 1 = n$, con lo que la condición III se modifica a:

III) $p(1) \geq p(2) \geq \ldots \geq p(n - 1) \geq p(n) \Rightarrow l(1) \leq l(2) \leq \ldots \leq l(n - 1) = l(n)$.

Supongase por otra parte que en el grupo de códigos con longitud $l(n)$ existen dos de ellos cuyos prefijos de orden $l(n) - 1$ son distintos; entonces, supuesta la condición de código prefijo, es posible eliminar el último dígito de dichos dos códigos con la consiguiente reducción de μ por lo que, si el código ha de ser óptimo al menos dos y hasta d de los códigos con longitud $l(n)$, deben tener prefijos de orden $l(n) - 1$ idénticos (dos porque los dos datos menos probables tienen longitud $l(n)$ y; hasta d porque solo hay d dígitos en el alfabeto de códigos los cuales podrían ser eliminados). Esto establece una cuarta condición para un código óptimo:

IV) Al menos 2 y no más de d de los datos con longitud de código $l(n)$ tienen códigos los cuales son parecidos excepto por sus dígitos finales.

Si por otra parte, existe una permutación de los símbolos de codificación de longitud d < $\ell(n)$ que no es utilizada como código para un dato o que no es prefijo de ningun otro código de otro dato, entonces utilizando dicha permutación como el código para el n-ésimo dato se obtiene una reducción en μ_ℓ.

Lo anterior se aplica a todas las permutaciones posibles p tales que d ≤ $\ell(p) \leq \ell(n)$ - 1 y; como el conjunto de permutaciónes de longitud $\ell(n)$ - 1 abarca en sus prefijos a todos los conjuntos de permutaciones de longitud menor que $\ell(n)$ - 1 una condición más para un código óptimo se enuncia como:

V) Cada sucesión posible de $\ell(n)$ - 1 dígitos debe ser usada ya sea como un código o prefijo de un dato o debe tener uno de sus prefijos usado como código de un dato.

En el razonamiento seguido por Huffman para deducir una serie de condiciones que debe cumplir un código óptimo, resalta la importancia que él le dió a la "condición de código prefijo" y debido a que las longitudes de los códigos así obtenidos minimizan la redundancia r del código, la cual está definida como la diferencia entre la longitud promedio del código μ_ℓ y la entropía μ del mensaje fuente:

$$r = \mu_\ell - \mu = \Sigma_i\, p(i)\, \ell(i) - \Sigma_i\, p(i)\, \left[-\log_d p(i)\right] \text{ en donde } i \in \{1,2,...,n\},$$

a un código que conste de un conjunto de códigos que cumplan dichas condiciónes se le llama "CODIGO PREFIJO OPTIMO".

Como se ha dicho con anterioridad, el esquema de codificación de Huffman ha sido ampliamente estudiado y hay quien dice que Huffman pudo o no tener en mente al desarrollarlo la estructura de árbol. El hecho es que si se cuenta con un alfabeto de códigos con cardinalidad igual a d, un código prefijo puede ser representado por un árbol d-ario etiquetado (un árbol d-ario es aquel en el que cada nodo puede tener 0,1, ..., d - 1 o d nodos hijos), en donde a cada dato posible en el mensaje de entrada le corresponde un nodo hoja, y el código del dato es la secuencia de etiquetas dadas por la ruta de la raíz a la hoja correspondiente.

Antes de dar un método general para cualquier cantidad d de símbolos de codificación, Huffman consideró el caso binario (d = 2 y alfabeto de códigos = {0,1}).

La condición III implica que los dos datos menos probables (n y n - 1) tendrán códigos con igual longitud $\ell(n)$ y; por IV éstos diferirán sólo en su dígito final, por lo que, y para llevar una secuencia lógica, al código del dato menos probable se le asignará como dígito final el 1 y al otro el 0. Una vez hecho lo anterior y debido a que el prefijo de orden $\ell(n)$ - 1 es el mismo para ambos códigos se pueden ver a ambos datos como uno solo compuesto, cuya probabilidad es la suma de

probabilidades y código el prefijo común de ambos. Con lo que se tiene un mensaje de datos auxiliar, el cual además de los datos originales contiene el dato compuesto en reemplazo por los dos a partir de los cuales fué creado, teniendo así un dato menos que el mensaje anterior.

El proceso anterior es llevado a cabo sobre cada mensaje de datos auxiliar nuevo hasta que el último mensaje auxiliar contenga un solo dato compuesto cuya probabilidad es 1.

Si bien Huffman pudo no tener en mente la estructura de árbol al desarrollar el algoritmo anterior es muy probable que sí haya tenido en mente el caracter probabilístico de los códigos, al estar éstos justamente en función de una distribución de probabilidad.

Dado que la entropía de un dato i con probabilidad $p(i)$ está dada por $e(i) = -\log_d p(i)$, si $p(i) = 1$, $e(i) = 0$, es decir, un dato con probabilidad 1 no comunica información, razón por la cual no vale la pena asignarle un código.

Entonces, sabiendo que la función de distribución acumulativa F(x) tiende a 1 cuando x tiende a infinito, es natural partir de los datos menos probables, acumular los requerimientos de codificación y continuar con los siguientes datos menos probables, hasta encontrar de esta forma que la función de distribución acumulativa ha alcanzado el 1, caso en el cual el dato (que en realidad son todos) aparecerá con certeza y no se le asigna un código.

Si se aplica el proceso dado al ejemplo anterior y se supone que el conjunto de datos posibles es {A, E, I, O, U} (por lo que n=5) se tienen los siguientes resultados:

	FRECUENCIAS DE LOS DATOS EN EL MENSAJE					*ASIGNACION DE CODIGOS (EMPEZANDO SIEMPRE POR EL ULTIMO DIGITO)*				
	MENSAJE	*MENSAJES AUXILIARES*				*EN LOS DIFERENTES ESTADOS DEL PROCESO*				
DATO	*ORIGINAL*	*1o*	*2o*	*3o*	*4o*	*0-ésimo*	*1o*	*2o*	*3o*	*CODIGOS*
A	*19*	*19*	*19*	*19*	*52* → *33*				*0*	
									1	*1*
					33			*0*	*00*	
								1	*01*	
				18	*18*					
			15	*15*						
E	*10*	*10*	*15*				*0*	*00*	*000*	*000*
I	*8*	*8*	*10*				*1*	*01*	*001*	*001*
O	*3*	*7*				*0*	*0*	*10*	*010*	*010*
U	*7*					*1*	*1*	*11*	*011*	*011*

En este punto vale la pena recordar que es prácticamente una utopía el disponer de las probabilidades de los datos por lo que, para efectos prácticos, se da una aproximación a éstas a partir de las frecuencias de cada dato en el mensaje.

A partir de los códigos obtenidos se puede construir una tabla similar a la construida para el algoritmo de Shannon-Fano:

DATO	FREC.	Cont. de Inf. en bits $-LOG_2(FREC./52)$	Bits de Inf. para el dato en el mensaje dados por su entropía	Cont. de Inf. por HUFF en bits	Bits de Inf. para el dato por HUFF
A	19	1.4525122	27.597732	1	19
E	10	2.3785116	23.785116	3	30
I	8	2.7004397	21.603518	3	24
O	8	2.7004397	21.603518	3	24
U	7	2.8930848	20.251594	3	21
	TD = 52		EM = 114.84148		EHUFF=118

en donde las diferencias con la anterior tabla se dan en las dos últimas columnas, es decir, cambian las longitudes de los códigos asociados a las letras A, E e I, disminuyendo la longitud del código asociado a A y aumentando las longitudes de los códigos asociados a E e I, en una forma tal que la longitud promedio del mensaje μ_l disminuye en 1 (de 119 pasa a 118), quedando de manifiesto, en este caso, la superioridad del esquema de codificación de Huffman sobre el de Shannon-Fano.

Una característica importante de este esquema de codificación es que, dado que al combinar dos datos se les asigna un dígito a cada uno de ellos, la longitud de un dato original (es decir, un dato en el mensaje original) es igual al número de veces que dicho dato, o uno compuesto del cual forma parte, es combinado. Así por ejemplo, el dato con frecuencia 7 es combinado primero con el dato con frecuencia 8 para formar un dato compuesto con frecuencia 15, el cual a su vez es combinado con el dato con frecuencia 16 para formar una vez más otro dato compuesto con frecuencia 33, el cual es combinado para formar el dato compuesto final. Por lo que el dato con frecuencia 7 tiene longitud igual a 3.

En la tabla anterior se puede observar que los datos I y O tienen la misma frecuencia por lo que, para obtener el primer mensaje auxiliar se pudieron haber tomado las frecuencias de los datos I y U, con lo cual, el único cambio hubiera sido en que el código asociado a I se le hubiera asociado a O y viceversa, sin afectar con ésto la longitud promedio del código. De hecho, puede ser que en cualquier estado del proceso se presente la situación de que hay más de un par de datos que son los menos probables y; la decisión de tomar un par específico provocará que los demás datos candidatos pasen a formar parte del siguiente mensaje auxiliar, nuevamente como candidatos.

Este nuevo mensaje auxiliar como ya se vio tiene n - 1 datos (si n es el número de datos en el mensaje auxiliar anterior); y si se han de acumular los requerimientos de codificación, para este nuevo mensaje auxiliar se ha de cumplir que:

$$p(1_{aux}) \geq p(2_{aux}) \geq \ldots \geq p(n_{aux} - 1) \geq p(n_{aux}) \Rightarrow l(1_{aux}) \leq l(2_{aux}) \leq \ldots \leq l(n_{aux} - 1) = l(n_{aux})$$

en donde (1_{aux}, 2_{aux}, ... , n_{aux} - 1, n_{aux}) es una reordenación de n - 2 de los datos del mensaje anterior y el dato compuesto a partir de los datos n - 1 y n del mismo mensaje y; donde al menos el último dato, es decir n_{aux} , es original del mensaje anterior (supuesto que en el mensaje anterior hubo más de un par de datos menos probables), y es el dato n - 2, por lo cual tendra longitud $l(n_{aux}) = l(n - 2) \leq l(n - 1) = l(n)$. En el ejemplo anterior al pasar del estado 0 al 1 el dato n - 2 es la letra I, la cual concuerda con el último dato n_{aux} en el primer mensaje de datos auxiliar y; al término del proceso $3 = l(I) = l(n_{aux}) = l(n - 2) \leq l(n - 1) = l(n) = l(U) = 3$. Por tanto, al tener más de una alternativa para elegir los dos datos menos probables en un mensaje cualquiera, la elección de dos cualesquiera de éstos provocará que los demás datos candidatos tengan longitud a lo más igual a la longitud de los datos elegidos.

Una situación como la descrita se presentará cuando como en el ejemplo anterior sólo haya un dato menos probable y al menos dos con igual probabilidad que le siguen, o cuando hay varios datos con igual probabilidad, la cual, es la menor.

En el primer caso al menos el dato n - 2 (podrían también ser los datos n - 3, n - 4, ...) será tal que $p(n - 2) = p(n - 1)$, en el segundo caso, al menos $p(n - 2) = p(n - 1) = p(n)$ y en cualquiera de los dos, al menos $l(n - 2)$ podría variar, pero como ya se vio, dos datos con la misma probabilidad pueden diferir en longitud y eso no afectará la longitud promedio del mensaje μ_l.

De lo anterior se concluye que al tener más de una alternativa para elegir los dos datos menos probables en cualquier estado del proceso, cualquiera de dichas elecciones puede hacerse y esto no afectará la longitud promedio del código, por lo que el procedimiento descrito siempre generará códigos los cuales, independientemente de sus longitudes, minimizarán la redundancia r del código, siendo así, un procedimiento suficiente para establecer un código binario óptimo.

Al plantear su esquema de codificación Huffman lo encontró análogo a las señales dejadas por un insecto de agua en cada conjunción de riachuelos que está en su camino al viajar diariamente flujo abajo. Para que el insecto pueda regresar flujo arriba debe hacer uso de dichas señales. En donde la confluencia de dos riachuelos en uno solo, es similar a la composición de dos datos en uno.

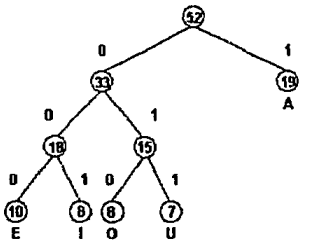
De la misma forma este esquema de codificación es similar al recorrido a través de un árbol binario etiquetado; y en este caso un nodo padre es el compuesto por sus dos nodos hijos (los dos hijos confluyen o componen al padre) y las etiquetas encontradas en el recorrido forman el código.

De acuerdo a esta última analogía y al método dado por Huffman el árbol binario es tal que se construye de abajo hacia arriba, es decir, a partir de los nodos hoja hasta llegar a la raíz. Por lo que se ha dado el siguiente algoritmo para construir el árbol de Huffman.

ALGORITMO DE HUFFMAN

PASO 1 A cada posible dato en el mensaje de entrada asociarle un nodo con peso igual a su probabilidad o frecuencia en el mensaje y llamar a este conjunto de nodos el conjunto de nodos libres.
PASO 2 Localizar en el conjunto de nodos libres los dos nodos libres con menor peso.
PASO 3 Crear el nodo padre para esos dos nodos y darle un peso igual a la suma de los pesos de éstos.
PASO 4 Etiquetar arbitrariamente con 0 y 1 a las dos ramas que van del padre a sus hijos.
PASO 5 Remover del conjunto de nodos libres a los dos anteriores e incluir al padre.
PASO 6 Repetir los pasos 2 a 5 hasta que el conjunto de nodos libres conste de un solo elemento, el cual será la raíz del árbol.

Aplicando el algoritmo de Huffman al ejemplo se obtiene el árbol de Huffman siguiente:



El código para un dato se obtiene recorriendo el árbol de la raíz a la hoja correspondiente y tomando nota de las etiquetas en la ruta. Así, se tiene la siguiente tabla de códigos:

DATO	CODIGO
A	1
E	000
I	001
O	010
U	011

a la cual se había llegado ya con anterioridad.

En el árbol anterior se puede observar que las etiquetas pueden ser cambiadas arbitrariamente sin afectar con ésto la longitud de los códigos (y con ésto la longitud promedio del código μ [.]), los cuales por otra parte seguirán también

cumpliendo los requerimientos impuestos para un código prefijo óptimo, por lo que se tienen 2^4 diferentes formas de asignar códigos a los datos del ejemplo. En general, si el mensaje de entrada tiene n posibles datos, existirán 2^{n-1} diferentes formas de asignar códigos a éstos.

Por lo anterior es que en el PASO 4 del algoritmo de Huffman no se da una regla para asignar etiquetas a las ramas.

Por otro lado, de la tabla se puede observar que la longitud del dato menos frecuente es 3, por lo que la condición V se enunciaría en este caso como: Cada sucesión posible de dos dígitos debe ser usada ya sea como un código o prefijo de un código de un dato o debe tener uno de sus prefijos usado como el código de un dato. En este caso dichas sucesiónes son 00,01,10 y 11; en donde 00 y 01 son prefijos y el prefijo común 1 de 10 y 11 es utilizado como código.

Debido a que al formar un dato compuesto se utilizan exactamente dos datos, al término del proceso, la raíz del árbol de Huffman siempre tendrá dos nodos hijos, cada uno de los cuales a su vez (si es que tiene) tendrá dos hijos, etc., lo que garantiza que todas las permutaciónes p de longitud $l(p) \leq l(n) - 1$ serán utilizadas (ya sea como el prefijo de o un código de un dato o alguno de sus prefijos será el código de un dato), que es justamente la condición V para un código prefijo óptimo.

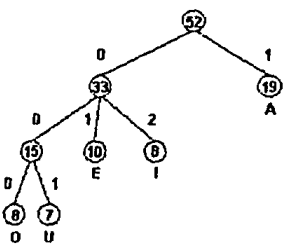
Y es precisamente esta última condición la que establece la diferencia entre el caso binario (d = 2) y el caso general (d ≥ 2) en donde, para garantizar que todas las permutaciónes p de longitud $l(p) \leq l(n) - 1$ serán utilizadas ya sea como el prefijo de o un código de un dato o alguno de sus prefijos será el código de un dato, en cada estado del proceso (con la posible excepción del 0-ésimo, ya que al empezar puede ser que haya menos de d datos) son utilizados d datos para obtener uno compuesto.

Siguiendo con el ejemplo y considerando el caso ternario (d = 3) se obtienen los siguientes resultados:

I) Utilizando dos dígitos en el 0-ésimo estado para formar uno compuesto.

	FRECUENCIAS DE LOS DATOS EN EL MENSAJE				ASIGNACION DE CODIGOS (EMPEZANDO SIEMPRE POR EL ULTIMO DIGITO)			
	MENSAJE	MENSAJES AUXILIARES			EN LOS DIFERENTES ESTADOS DEL PROCESO			
DATO	ORIGINAL	1o	2o	3o	0-ésimo	1o	2o	CODIGOS
				→ 52				
			→ 33				0	
A	19	19	19				1	1
		→ 15				0	00	
E	10	10				1	01	01
I	8	8				2	02	02
O	8				0	00	000	000
U	7				1	01	001	001

cuyo árbol ternario correspondiente es:



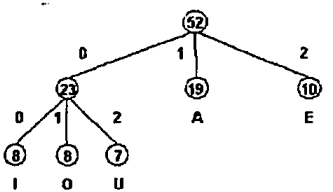
y cuya tabla que nos indica la entropía del mensaje dada por el modelo probabilístico es:

DATO	FREC.	Cont. de Inf. en base 3 $-LOG_3(FREC./52)$	Dígitos de Inf. para el dato en el mensaje dados por su entropía	Cont. de Inf. por HUFF en base 3 HUFF	Dígitos de Inf. para el dato por HUFF
A	19	0.9164331	17.412230	1	19
E	10	1.5006738	15.006738	2	20
I	8	1.7037878	13.630302	2	16
O	8	1.7037878	13.630302	3	24
U	7	1.8253333	12.777333	3	21
	TD = 52		EM = 72.456905		EHUFF=100

II) Utilizando tres dígitos en el estado 0-ésimo para formar uno compuesto.

	FRECUENCIAS DE LOS DATOS EN EL MENSAJE			*ASIGNACION DE CODIGOS (EMPEZANDO SIEMPRE POR EL ULTIMO DIGITO)*		
DATO	*MENSAJE ORIGINAL*	*MENSAJES AUXILIARES* 1o	2o	*EN LOS DIFERENTES ESTADOS DEL PROCESO* 0-ésimo	1o	*CODIGOS*
			→ 52			
		→ 23			0	
A	19	19			1	1
E	10	10			2	2
I	8			0	00	00
O	8			1	01	01
U	7			2	02	02

cuyo árbol correspondiente es:



y cuya tabla correspondiente es:

DATO	FREC.	Cont. de Inf. en base 3 $-LOG_3(FREC./52)$	Dígitos de Inf. para el dato en el mensaje dados por su entropía	Cont. de Inf. por HUFF en base 3 HUFF	Dígitos de Inf. para el dato por HUFF
A	19	0.9164331	17.412230	1	19
E	10	1.5006738	15.006738	1	10
I	8	1.7037878	13.630302	2	16
O	8	1.7037878	13.630302	2	16
U	7	1.8253333	12.777333	2	14
	TD = 52		EM = 72.456905		EHUFF = 75

Si se supone que el conjunto de caracteres ASCII es codificado con dígitos ternarios, se tendrá que un caracter de tal conjunto se podrá guardar en seis de ellos; y si adicionalmente se supone que los datos forman parte de tal conjunto, el mensaje ocupará TD X 6 = 52 X 6 = 312 dígitos ternarios sin codificación, requiriendo el caso I de solo 100 y el II de 75, obteniendo las tasas de compresión del 67.95 % (1 - 100 / 312) y del 75.96 % (1 - 75 / 312) para los casos I y II respectivamente, siendo la óptima del 76.78 % (1 - 72.456905 / 312).

En esta ocasión resulto ser mejor el caso II (es decir, tomar d = 3 datos en el estado 0-ésimo para formar uno compuesto) aunque no siempre sucede así.

Si se observa detenidamente cada uno de los árboles de Huffman generados hasta aquí, se notará que todos ellos son tales que si se recorren los nodos (empezando por el raíz) de arriba hacia abajo y de izquierda a derecha, los pesos formarán una sucesión no creciente de números reales.

Lo anterior no es coincidencia. De hecho un árbol de Huffman cumple la propiedad de Sibling, la cual establece que: Los nodos (excepto el raíz) pueden ser listados en orden de peso no creciente de modo que $\forall i \; 1 \leq i \leq c \; (c \in Z)$ los nodos *id*, *id* - 1, ... , *id* - d + 1 son hermanos, supuesto que los nodos excepto el raíz han sido numerados en orden de arriba hacia abajo y de izquierda a derecha.

Como se ha dicho en repetidas ocasiones el esquema de codificación de Huffman ha sido ampliamente estudiado, lo que ha llevado a una serie de resultados, algunos de los cuales (como el anterior) son útiles para propósitos de implementación [Mark92, Stor92].

Toda esa serie de resultados forman por sí solos un área de estudio, la cual por ser extensa queda fuera de los propósitos de este trabajo de tesis.

ESQUEMAS DE DICCIONARIO

En muchas situaciones de comunicación digital y procesamiento de datos, las cadenas de datos encontradas presentan regularidades estructurales o están sujetas a ciertas restricciónes, conllevando así a una compresión potencial de dichos datos. Por lo que el problema de compresión de datos, en general, es resuelto en dos etapas: primero son identificadas las regularidades o restricciónes de la fuente a comprimir y después se diseña un esquema de codificación el cual, sujeto a criterios de desarrollo, comprimirá mejor la fuente.

Una vez determinados los parámetros relevantes de la fuente, el problema se reduce al de codificación de mínima redundancia. Tal es el caso del esquema de codificación de Huffman, en donde las regularidades o restricciones son reflejadas por la distribución de probabilidad asociada a la fuente.

Estrictamente hablando la afirmación anterior implica un conocimiento a-priori de la fuente (su distribución de probabilidad o una aproximación a ella), el cual generalmente sólo se obtiene analizándola (en un paso adicional) antes de llevar a cabo la codificación propiamente dicha.

Este problema no es exclusivo del esquema de codificación de Huffman, sino que es común a todos aquellos que utilizan modelado estadístico, los cuáles para derivar las estadísticas de la fuente, implementan pasos adicionales en el algoritmo para desarrollar el modelo, por lo que la tasa de compresión esta en función de que tan bien (o mal) esté el algoritmo desarrollando el modelo.

Más aún, las pruebas estadísticas pueden ser imposibles o desconfiables, con lo cual el problema llega a ser considerablemente más complicado, caso en el cual se debe acudir a ESQUEMAS DE CODIFICACION UNIVERSALES, en los que el proceso de codificación está asociado a un proceso de aprendizaje para las características variantes de la fuente.

Tales esquemas universales inevitablemente requieren un espacio de memoria mayor para trabajar, y generalmente emplean criterios de desarrollo que son apropiados para una amplia variedad de fuentes.

Entonces, por un esquema de codificación universal entenderemos un esquema que puede ser aplicado a prácticamente cualquier fuente de datos, ya que por regla general éste tipo de esquemas no requieren información a-priori de la fuente a codificar.

Por otra parte, debido a que hasta la década de los 70's la investigación en compresión de datos se centró en entropía, frecuencias de palabras y caracteres y otras facetas del modelado estadístico (habiendo menores incursiones en otras áreas de interés, tales como máquinas de estado finito y modelos lingüísticos), prácticamente no había esquemas de codificación universales.

Pero todo cambió cuando en mayo de 1977 Jacob Ziv y Abraham Lempel publicaron su artículo "A Universal Algorithm For Sequential Data Compression" [ZL77]. Ese artículo junto con su secuela de septiembre de 1978 "Compression of Individual Sequences via Variable-Rate Coding" [ZL78] motivaron un diluvio de investigaciones, algoritmos y programas sobre compresión basada en diccionario.

Los esquemas de codificación basados en diccionario utilizan un método completamente diferente para comprimir los datos. Estos no codifican símbolos aislados como cadenas de bits de longitud variable, sino que codifican cadenas de símbolos de longitud variable en códigos de longitud fija, en donde dichos códigos constituyen un índice o apuntador a una frase del diccionario.

Aunque si bien existen esquemas de diccionario basados en un diccionario estático, éstos normalmente no son de propósito general, son dependientes de la implementación y lo que es peor, otros métodos, tales como diccionarios deslizantes (Sliding Dictionary) o diccionarios dinámicos, los pueden superar (aún a diccionarios estáticos que han sido especialmente construidos para los datos que van a comprimir), razón por la cual muchos de los esquemas de diccionario más conocidos (y utilizados) son adaptativos.

Estos últimos, lejos de tener un diccionario completamente definido antes de iniciar el proceso de codificación, empiezan sin un tal diccionario o con uno default, el cual, conforme la codificación progresa es actualizado, adicionándole nuevas frases a ser utilizadas posteriormente.

Como hemos mencionado con anterioridad las cadenas de datos encontradas en muchas situaciónes de comunicación digital y procesamiento de datos muestran regularidades estructurales, y es precisamente éste tipo de regularidades las que Jacob Ziv y Abraham Lempel intentaron explotar en su algoritmo propuesto en 1977 mediante un esquema de compresión adaptativo basado en diccionario.

CODIFICACION LZ77

Si buscamos las raíces de, virtualmente, cualquier algoritmo de compresión basado en diccionario, las hallaremos en los trabajos de Ziv y Lempel [ZL77, ZL78]. Estos dos investigadores israelíes fueron quienes prácticamente inauguraron ésta rama de la teoría de la información en los 70's con la publicación de sus artículos del 77 y 78, los cuales describen técnicas de compresión que son referidas como LZ77 y LZ78.

En la técnica LZ77 el diccionario consiste de un conjunto de frases de longitud variable, pero acotada por un entero L_s predeterminado, halladas en una ventana móvil en el texto previamente procesado, razón por la que ésta técnica es conocida como de Sliding Window (ventana deslizante).

Para describir en forma precisa el algoritmo LZ77 necesitamos alguna preparación en forma de notación y definiciones:

- Considerar un alfabeto finito A de α símbolos, digamos $A = \{0, 1, \ldots, \alpha - 1\}$. Una cadena o palabra S de longitud $l(S) = k$ sobre A es un k-túple ordenado $S = s_1 s_2 \ldots s_k$ de símbolos de A.
- Para indicar una subcadena de una cadena S sobre A, la cual empieza en la posición i y finaliza en la posición j, escribimos $S(i, j)$. Cuando $i \leq j$, $S(i, j) = s_i s_{i+1} \ldots s_j$, pero cuando $i > j$, $S(i, j) = \Lambda$, la cadena nula de longutid $l(\Lambda) = 0$.
- La concatenación de dos cadenas Q y R forman una nueva cadena S = QR; si $l(Q) = k$ y $l(R) = m$, entonces $l(S) = k + m$, $Q = S(1,k)$ y $R = S(k + 1, k + m)$.
- Para cada j $0 \leq j \leq l(S)$, $S(1, j)$ es llamado un prefijo de S; $S(1, j)$ es prefijo propio de S si $j < l(S)$.
- Dado un prefijo propio $S(1, j)$ de una cadena S y un entero positivo i tal que $i \leq j$, $L(i)$ denota el entero no negativo más grande $l \leq l(S) - j$ tal que $S(i, i + l - j) = S(j + 1, j + l)$, y sea p la posición de $S(1, j)$ para la cual $L(p) = \max_i \{L(i)\}$ con $1 \leq i \leq j$. La subcadena $S(j + 1, j + L(p))$ de S, es llamada la extensión reproducible de $S(1, j)$ en S, y el entero p es llamado el apuntador de la reproducción.

Por ejemplo, si S = 00101011 y $j = 3$, entonces L(1) = 1 puesto que $S(j + 1, j + 1) = S(1, 1)$ pero $S(j + 1, j + 2) \neq S(1,2)$. Similarmente, L(2) = 4 y L(3) = 0. Así, S(3 + 1, 3 + 4) = 0101 es la extensión reproducible de S(1,3) = 001 en S con apuntador de la reproducción p = 2.

Ahora, si $S = s_1 s_2 s_3 \ldots$ denota la cadena de símbolos emitidos por la fuente, el esquema de codificación LZ77 analiza S en palabras sucesivas $S = S_1 S_2 S_3 \ldots$ y asigna un código C_i de longitud L_c fija a cada S_i de longitud l_i a lo más igual a un entero predeterminado L_s.

Además de los parámetros L_c y L_s es definido un tercer parámetro n, el cual es la longitud de un buffer que almacena los n símbolos más recientes emitidos por la fuente.

Para iniciar el proceso de codificación, asumimos que la salida S de la fuente fue precedida por una cadena Z de $n - L_s$ ceros, y almacenamos la cadena $B_1 = ZS(1, L_s)$ en el buffer. Si $S(1, j)$ es la extensión reproducible de Z en $ZS(1, L_s - 1)$, entonces $S_1 = S(1, j + 1)$ y $l_1 = j + 1$. Para determinar la próxima palabra fuente, removemos los primeros l_1 símbolos del buffer y liberamos en él los próximos l_1 símbolos de S para obtener la cadena $B_2 = B_1(l_1 + 1, n)S(L_s + 1, L_s + l_1)$. Ahora nos fijamos en la extensión reproducible E de $B_2(1, n - L_s)$, en $B_2(1, n - 1)$, y fijamos $S_2 = Es$, donde s es el próximo símbolo después de E en B_2. En general, si B_i denota la cadena de n símbolos fuentes almacenados en el buffer cuando nos disponemos a determinar la i-ésima palabra fuente S_i, los pasos sucesivos de la codificación pueden ser descritos como sigue:

·) Habiendo determinado B_i, $i \geq 1$, calcular $S_i = B_i(n - L_s + 1, n - L_s + l_i)$, donde el prefijo de longitud $l_i - 1$ de S_i es la extensión reproducible de $B_i(1, n - L_s)$ en $B_i(1, n - 1)$.

·) Si p_i es el apuntador de la reproducción usado para determinar S_i, entonces el código C_i para S_i está dado por $C_i = C_{i1}C_{i2}C_{i3}$, donde C_{i1} es la representación en base α de $p_i - 1$, C_{i2} es la representación en base α de $l_i - 1$ y C_{i3} es el último símbolo de S_i, es decir, el símbolo que ocupa la posición $n - L_s + l_i$ de B_i.

·) Para actualizar el contenido del buffer, remover los símbolos que ocupan las primeras l_i posiciones de éste mientras se liberan los próximos l_i símbolos de la fuente, para obtener $B_{i+1} = B_i(l_i + 1, n)S(h_i + 1, h_i + l_i)$, donde h_i es la posición de S ocupada por el último símbolo de B_i.

Para ilustrar el mecanismo del algoritmo consideremos la cadena temaria de entrada ($\alpha = 3$) S = 00101021021021202102120 0..., y un codificador con parametros $L_s = 9$ y $n = 18$.

Inicialmente, el buffer es cargado con $n - L_s = 9$ ceros, seguidos por los primeros $L_s = 9$ dígitos de S, a saber, $B_1 = 000000000001010210$. Tenemos que hallar el prefijo más largo $B_1(10, 9 + l_1 - 1)$ de $B_1(10, 17) = 00101021$ el cual concuerda con una subcadena de B_1 que empieza en la posición $p_1 \leq 9$ y entonces fijar $S_1 = B_1(10, 9 + l_1)$. En éste caso, la concordancia más larga es $B_1(10, 11) = 00$, y así $l_1 = 3$ y $S_1 = 001$. El apuntador p_1 para éste caso puede ser cualquier entero entre 1 y 9 , elegimos $p_1 = 9$.

La representación en base 3 de $p_1 - 1 = 8$ es $C_{11} = 22$, la de $l_1 - 1 = 2$ es $C_{12} = 02$ y puesto que C_{13} es igual al último símbolo de S_1, el código para S_1 es $C_1 = C_{11}C_{12}C_{13} = 22021$.

Para obtener la carga B_2 del buffer para el segundo paso, removemos los primeros $l_1 = 3$ dígitos de B_1 y liberamos los próximos $l_1 = 3$ dígitos S(10, 12) = 210 de la cadena S de entrada. Los detalles del algoritmo son tabulados abajo, donde los dígitos en negritas corresponden a las posiciones del apuntador de la reproducción (posiciones 1 a $n - L_s = 18 - 9 = 9$), las palabras fuente S_i son indicadas por la subcadena itálica y las extensiones reproducibles (de longitud $l_i - 1$) por la subcadena itálica negrita de la correspondiente carga del buffer B_i.

i	B_i	p_i	$L(p_i)$	$S_i(l_i)$	C_{i1}	C_{i2}	C_{i3}	C_i
1	B1 = 000000000*0*01010210	9	2	1	22	02	1	22021
2	B_2 = 00000000*1*0*10*210210	8	3	2	21	10	2	21102
3	B_3 = 0000101020*10210212*0	7	7	2	20	21	2	20212
4	B_4 = 2102102120*2102120*0	3	8	0	02	22	0	02220

Con lo que el mensaje codificado será C = $C_1C_2C_3C_4$... = 22021211022021202220...

En éste ejemplo la longitud L_c de cada C_i fue igual a 5 la cual pudimos haber determinado desde el principio puesto que está en función de n y L_a.

Para derivar la fórmula para obtener L_c consideremos la progresión geométrica de razón α, $P = \alpha^0+\alpha^1+\alpha^2+ \dots +\alpha^n$, entonces $\alpha P = \alpha^1+\alpha^2+\alpha^3+ \dots +\alpha^{n+1}$, con lo que $(\alpha - 1)P = \alpha P - P = \alpha^{n+1} - \alpha^0 = \alpha^{n+1} - 1$, es decir, $(\alpha - 1)(\alpha^0+\alpha^1+\alpha^2+ \dots +\alpha^n) = \alpha^{n+1} - 1$, lo que implica que $\alpha^{n+1} - 1$ es el mayor entero que puede ser representado con $\log_\alpha \alpha^{n+1} = n + 1$ dígitos α-arios (es decir, $(\alpha^{n+1} - 1)_{10} = [(\alpha - 1)(\alpha - 1)(\alpha - 1) \dots (\alpha - 1)]_\alpha$, donde $\alpha - 1$ se repite $n + 1$ veces).

Ahora, debido a que los apuntadores reproducibles p_i sólo pueden tomar valores enteros entre 1 y $n - L_a$, $P_i - 1$ tomará valores entre cero y $(n - L_a) - 1$, por lo que para representar $p_i - 1$ en base α necesitaremos de $\lceil \log_\alpha (n - L_a) \rceil$ dígitos α-arios (aquí $\lceil x \rceil$ es el menor entero mayor o igual que x), es decir, $l(p_i - 1)_\alpha) = l(C_{i1}) = \lceil \log_\alpha (n - L_a) \rceil$. De la misma forma y ya que el valor máximo que puede tomar $l_i - 1$ es $L_a - 1$, para representar $l_i - 1$ en base α necesitaremos de $\lceil \log_\alpha L_a \rceil$ dígitos α-arios, es decir, $l(l_i - 1)_\alpha) = l(C_{i2}) = \lceil \log_\alpha L_a \rceil$.

Por tanto, dado que $C_i = C_{i1}C_{i2}C_{i3}$, $l(C_i) = l(C_{i1}) + l(C_{i2}) + l(C_{i3}) = \lceil \log_\alpha (n - L_a) \rceil + \lceil \log_\alpha L_a \rceil + 1$ (recordando que $C_{i3} = S_i(l_i)$ y por tanto es un solo dígito, $l(C_{i3}) = 1$) para toda i. A ésta cantidad constante la hemos ya denotado por L_c, es decir, $L_c = \lceil \log_\alpha (n - L_a) \rceil + \lceil \log_\alpha L_a \rceil + 1$.

Para nuestro ejemplo: $Lc = \lceil \log_3 (18 - 9) \rceil + \lceil \log_3 9 \rceil + 1 = 2 + 2 + 1 = 5$, de lo cual ya nos habíamos percatado.

Por otra parte, la decodificación puede ser desarrollada invirtiendo el proceso de codificación. Aquí empleamos un buffer de longitud $n - L_a$ para almacenar los más recientes símbolos fuentes decodificados. Inicialmente el buffer es cargado con $n - L_a$ ceros. Si $D_i = d_1d_2 \dots d_{n-La}$ denota el contenido del buffer después que C_i ha sido decodificado en S_i, entonces $S_i = D_i(n - L_a - l_i + 1, n - L_a)$, donde $l_i = l(S_i)$ y donde D_{i+1} puede ser obtenido de D_i y C_{i+1} como sigue:

Determinar $p_{i+1} - 1$ y $l_{i+1} - 1$ de los primeros $\lceil \log_\alpha (n - L_a) \rceil$ y los próximos $\lceil \log_\alpha L_a \rceil$ símbolos de C_{i+1}. Luego aplicar $l_{i+1} - 1$ shifts mientras se libera el contenido de la dirección p_{i+1} en la dirección $n - L_a$. El primero de esos shifts cambiará el contenido del buffer de D_i a $D_{i(1)} = d_2d_3 \dots d_{n - La}d_{pi+1} = d_{1,(1)}d_{2,(1)} \dots d_{n - La,(1)}$. Similarmente, si $j \le l_{i+1} - 1$, el j-

ésimo shift transformará $D_{p_i - 1} = d_{1,(i-1)}d_{2,(i-1)}...d_{n - Ls,(i-1)}$ en $D_{k(i)} = d_{2,(i-1)}d_{3,(i-1)}...d_{n - Ls,(i-1)}d_{pi + 1,(i-1)} = d_{1,(i)}d_{2,(i)}...d_{n - Ls,(i)}$, después de los primeros $l_i - 1$ shifts, aplicar un shift más mientras se libera el último símbolo de C_{i+1} en la dirección $n - L_s$ del buffer. Entonces, la carga del buffer resultante contiene S_{i+1} en sus últimas $l_{i+1} = l(S_{i+1})$ posiciones. Nuevamente con el objeto de aclarar el algoritmo de decodificación apliquémoslo a nuestro anterior ejemplo:

Al codificar habíamos llegado a la secuencia C = $C_1C_2C_3C_4...$ = 22021211022021202220... Entonces, el primer contenido D_0 del buffer de longitud $n - L_s = 18 - 9 = 9$ son puros ceros, es decir, D_0 = 000000000 y a partir de éste y de C_1 determinamos D_1 como sigue: Leemos los primeros $\lceil \log_\alpha (n - L_s) \rceil = \lceil \log_3 (18 - 9) \rceil = \lceil 2 \rceil = 2$ dígitos de C_1 = 22021 en $p_1 - 1$ y los siguientes $\lceil \log_\alpha L_s \rceil = \lceil \log_3 9 \rceil = \lceil 2 \rceil = 2$ dígitos de C_1 en $l_1 - 1$, es decir, $p_1 - 1 = 22_3 = 8_{10}$ y $l_1 - 1 = 02_3 = 2_{10}$. Luego aplicamos $l_1 - 1 = 2$ shifts al tiempo de cargar el contenido de la localidad $p_1 = 9$ en la localidad $n - L_s = 18 - 9 = 9$ del mismo buffer, con lo que obtenemos después del primer shift, $D_{0,(1)}$ = 000000000 y después del segundo shift, $D_{0,(2)}$ = 000000000. Por último, aplicamos un shift más y liberamos el último dígito de C_1 en la localidad $n - L_s = 9$ del buffer, con lo que $D_{0,(3)} = D_1$ = 000000001.

Hemos así obtenido el siguiente contenido D_1 del buffer, el cual contiene en sus últimas $l_1 = 3$ localidades a S_1, es decir, D(7, 9) = S_1 = 001 el cual debe ser mandado como parte de la fuente decodificada a la salida.

Nuevamente los detalles del algoritmo son mostrados en forma tabular a continuación, en donde las subcadenas itálicas de cada D_i calculada al final de una iteración son las S_i's correspondientes.

i	D_i	C_{i+1}	$p_{i+1} - 1$	$l_{i+1} - 1$	k	$D_{k(k)}$	S_{i+1}
0	000000000	22021	8	2	1	000000000	
					2	000000000	
					3	00000000*1*	001
1	000000001	21102	7	3	1	000000010	
					2	000000101	
					3	000001010	
					4	00000*1102*	0102
2	000010102	20212	6	7	1	000101021	
					2	001010210	
					3	010102102	
					4	101021021	
					5	010210210	
					6	102102102	
					7	021021021	
					8	*210210212*	10210212
3	210210212	02220	2	8	1	102102120	
					2	021021202	
					3	210212021	
					4	102120210	
					5	021202102	
					6	212021021	
					7	120210212	
					8	202102120	
					9	*021021200*	021021200

Con ésto hemos reconstruido la fuente original S como la secuencia $S = S_1S_2S_3S_4...$

Si analizamos los códigos obtenidos, nos percataremos de que en el $C_4 = 02220$ la parte correspondiente a la longitud de la cadena ($C_{42} = 22_3$) tomo el valor más grande posible al codificar la secuencia 021021200 y si observamos la secuencia S de entrada, nos daremos cuenta de que antes de la anterior secuencia ocurrio la secuencia 02102120.

Lo anterior ilustra una propiedad importante del algoritmo LZ77, que éste explota los "parecidos ocurridos recientemente". Aunque no así los ocurridos no tan recientemente, ya que debido a su pequeña ventana y a su mecánica misma, una misma cadena que aparece en, digamos, 2 páginas diferentes no tiene por qué en ambas ocasiones ocupar la misma posición en la ventana, con lo que normalmente se generan dos entradas diferentes en el diccionario para guardar partes de una misma cadena que aparece en dos lugares no cercanos en la secuencia S de entrada.

Es seguro que los autores del algoritmo anterior hayan estado conscientes de ésta deficiencia y al año siguiente dieron una variante mejorada de él [ZL78], al tiempo de utilizarla como una demostración constructiva de un teorema que establece la existencia de un codificador de una cierta clase que cumple ciertas características de desarrollo, el cual es conocido como LZ78.

CODIFICACION LZ78

Los códigos LZ (como normalmente son referidos los esquemas derivados por Lempel y Ziv) caen en una clase llamada modelado de contexto finito en la cual se utiliza el contexto de los más recientes datos de entrada para codificar el dato actual. Esta clase a su vez es un caso particular de una clase más general, conocida como modelado de estado finito (o modelado Markov), la cual permite explotar las características de la entrada que no pueden ser representadas en modelos de contexto finito. Por ejemplo, un modelo de estado finito puede representar información tal como "en un archivo hay un coro cada 4 caracteres" o "cada secuencia de a's tiene longitud par".

Como se puede intuir, si el modelado de contexto hoy en día es un nuevo enfoque del modelado estadístico muy prometedor, aún lo es más el modelado de estado finito, aunque los investigadores aún no han utilizado de manera exitosa esa creciente potencialidad. El desarrollo de métodos para construir modelos de estado finito dinámicamente es un problema abierto, cuya solución tiene un apreciable valor.

En su artículo "Compression of Individual Sequences via Variable-Rate Coding" [ZL78], publicado en septiembre de 1978, Ziv y Lempel establecieron dos teoremas para la clase de codificadores recuperables que utilizan modelado de estado finito.

El primer teorema es un teorema asintótico que establece una cota inferior para la compresibilidad $\rho_{E(s)}(X_{1,n})$ de una cadena $X_{1,n} = X_1X_2...X_n$ en una secuencia X de entrada, sobre la clase E(s) de los codificadores recuperables de estado finito $\leq s$, en donde dicha compresibilidad es definida como el mínimo sobre toda la clase de estos codificadores de la tasa de compresión para la cadena $X_{1,n}$, es decir, $\rho_{E(s)}(X_{1,n}) = min_E \{\rho_E(X_{1,n})\}$ con $E \in E(s)$.

El concepto de compresibilidad como es definido por Ziv y Lempel en su artículo del '78 parece jugar un papel análogo al de entropía en la teoría de la información clásica.

El segundo teorema (que es el que a nosotros nos interesa) demuestra utilizando una variante del algoritmo LZ77 (al que después se conocería por LZ78), la existencia de un esquema de codificación recuperable de estado finito universal, asintóticamente óptimo, bajo el cual la tasa de compresión obtenible para X tiende en el límite a la compresibilidad $\rho(X)$ de X, para cada X.

Al utilizar el algoritmo LZ78 tanto el codificador como el decodificador emplezan con un diccionario que sólo contiene la cadena nula Λ de longitud cero. Entonces, al leer cada nuevo dato de entrada, éste es concatenado a la cadena actual, proceso el cual continúa en tanto que la cadena actual (resultante de la concatenación) corresponda a una frase en el diccionario.

Eventualmente la cadena actual no será parte del diccionario, y es en éste punto donde los esquemas LZ77 y LZ78 toman acciones diferentes.

Si recordamos que la cadena actual está formada por la concatenación de una cadena en el diccionario, a la cual le llamaremos la última concordancia, y el último dato leído, en éste punto el algoritmo LZ77 emite como código el apuntador en el diccionario a la última concordancia, su longitud y el último dato leído.

Sin embargo, el algoritmo LZ78 en un paso adicional suma la cadena actual al diccionario, con lo que la próxima vez que aparezca ésta podrá ser utilizada para implementar una frase más larga.

Para llevar a cabo la codificación de un bloque $X_{1,n}$ (recordemos que $X_{1,n}$ está definido como la cadena $X_1X_2...X_n$) de una secuencia X de entrada, éste es analizado en segmentos o palabras, de acuerdo a un así llamado procedimiento de análisis incremental. Dicho análisis es indicado como $X_{1,n} = X_{n_0+1,n_1}X_{n_1+1,n_2}X_{n_2+1,n_3}...X_{n_p+1,n_{p+1}}$ donde $n_0 = 0$ y $n_{p+1} = n$ y es llamado incremental si las primeras p palabras $X_{n_{j-1}+1,n_j}$ $1 \leq j \leq p$, son todas distintas y si para toda $j = 1, 2, ..., p + 1$ cuando $n_j - n_{j-1} > 1$ existe un entero positivo $i < j$ tal que $X_{n_{i-1}+1,n_i} = X_{n_{j-1}+1,n_j-1}$.

Por ejemplo, consideremos las secuencias binarias X(1),X(2) y X(3) de longitudes $1(2^1)$, $2(2^2)$ y $3(2^3)$ respectivamente que listan en orden lexicográfico todas las 2^1, 2^2 y 2^3 palabras binarias de longitudes 1, 2 y 3 respectivamente, es decir, X(1) = 01, X(2) = 00011011 y X(3) = 000001010011100101110111, entonces dada $X_{1,\ 34}$ = X(1)X(2)X(3) = 0100011011000000101001110010111 0 un análisis incremental de $X_{1,\ 34}$ es $X_{1,\ 34}$ = 0,1,00,01,10,11,000,001,010,011,100,101,110,111.

Aquí $n_0 = 0$, $n_1 = 1$, $n_2 = 2$, $n_3 = 4$, $n_4 = 6$, $n_5 = 8$, $n_6 = 10$, $n_7 = 13$, $n_8 = 16$, $n_9 = 19$, $n_{10} = 22$, $n_{11} = 25$, $n_{12} = 28$, $n_{13} = 31$ y $n_{14} = 34 = n_{p+1}$ y se verifica que las primeras p = 13 palabras son distintas. Además , si por ejemplo $j = 3$, $n_j - n_{j-1} = n_3 - n_2 = 4 - 2 = 2 > 1$ y para $i = 1$ $X_{n_{i-1}+1,n_i} = X_{n_{1-1}+1,n_1} = X_{n_0+1,n_1} = X_{0+1,1} = X_{1,1} = 0 = X_{3,3} = X_{2+1,\ 4-1} = X_{n_{3-1}+1,\ n_3-1} = X_{n_{j-1}+1,n_j-1}$. Esta última igualdad muestra una característica que es común a todos los análisis incrementales: Para cada palabra de longitud $l > 1$, su prefijo de longitud $l - 1$ puede ser hallado como una palabra anterior del análisis.

Una vez provistos de ésta terminología y si definimos $X_{n_{-1}+1,n_0} = \Lambda$ como la palabra nula de longitud cero y ya que para cualquier palabra X $X = \Lambda X$, convendremos en que Λ es siempre la palabra con que inicia cualquier análisis incremental. Por otra parte, dada una palabra w, denotaremos por d(w) a la palabra obtenida al remover el último dato de w, por lo que la característica antes mencionada nos dice que para $j = 1, 2, ..., p + 1$ existe un único entero no negativo $\Pi(j) = i < j$, tal que $d(X_{n_{j-1}+1,n_j}) = X_{n_{i-1}+1,n_i}$.

El esquema LZ78 desarrolla un análisis incremental sobre cada bloque $X_{1,\ n}$ de la secuencia X de entrada de longitud n predeterminada, y codifica las palabras utilizadas por dicho análisis.

Lo anterior es llevado a cabo de manera secuencial determinando la j-ésima ($1 \leq j \leq p + 1$) palabra del análisis incremental de la siguiente manera:

-Tomar $n_j > n_{j-1}$ como el entero mayor que no excede a n para el cual $d(X_{n_{j-1}+1,n_j})$ es igual a alguna palabra anterior; por ejemplo, $X_{n_{j-1}+1,n_j}$ y calculemos $\Pi(j) = i$ (por ejemplo para $j = 1$, $n_1 = 1$, $X_{1,1} = X$, $d(X_1) = \Lambda$ y $\Pi(1) = 0$).

- Habiendo determinado $X_{n_{j-1}+1,n_j}$, el entero $I(X_{n_{j-1}+1,n_j}) = \Pi(j)\alpha + I_A(X_{n_j})$ es codificado en su representación en base dos. Aquí I_A es un mapeo predeterminado del alfabeto A de entrada sobre el conjunto de enteros 0 a α - 1.

Puesto que $0 \leq \Pi(j) \leq j - 1$, $0 \leq I(X_{n_{j-1}+1,n_j}) \leq (j-1)\alpha + \alpha - 1 = j\alpha - 1$, por lo que el número de bits requeridos para codificar la j-ésima palabra es $L_j = \lceil \log(j\alpha) \rceil$, el menor entero no más chico que $\log(j\alpha)$.

Si consideramos nuevamente el segmento $X_{1,34}$ = 0100011011000001010011100101110111, y si definimos el mapeo I_A como $I_A(X_{n_j}) = X_{n_j}$, la función identidad, entonces al desarrollar el algoritmo tenemos los siguientes resultados tabulares:

j	n_j	$d(X_{n_{j-1}+1,n_j})$	$\Pi(j) = i$	$I_A(X_{n_j})$	$I(X_{n_{j-1}+1,n_j})$
1	1	Λ	0	0	0
2	2	Λ	1	1	112
3	4	0	1	0	102
4	7	01	4	1	10012
5	11	011	5	0	10102
6	14	00	3	0	110
7	18	0	1	1	11
8	19	01	7	0	1110
9	23	011	4	1	1001
10	26	00	3	1	111
11	31	0111	9	0	10010
12	33	1	2	1	101
13	34	Λ	2	1	101

Siendo el código de salida 011101001101011011111010011111001010110 1.

El proceso de decodificación toma como entrada una secuencia $b = b_1 b_2 \ldots$ binaria, la cual es analizada en los códigos $b_{k_0+1,k_1}, b_{k_1+1,k_2}, \ldots$, que son descifrados en las frases originales de acuerdo al siguiente procedimiento:

- Inicialmente, fijar $j = 0$, $k_j = 0$ y $n_j = 0$.
- Dados los valores actuales de j, k_j y $n_j < n$ llevar a cabo lo siguiente:

i) Calcular $k_{j+1} = k_j + \lceil \log((j+1)\alpha) \rceil$.

ii) Tomar $l(X_{n_j+1}, n_{j+1})$ el entero cuya representación en base dos está dada por b_{k_j+1}, k_{j+1}, y determinar los enteros i y r no negativos que satisfacen $l(X_{n_j+1}, n_{j+1}) = i\alpha + r$, $0 \le r \le \alpha - 1$.

iii) Calcular $a = I_A^{-1}(r)$, y si $n_j + n_i - n_{i-1} + 1 \ge n$, calcular $n_{j+1} = n$, tomar $X_{n_j+1, n}$ la palabra formada por los primeros $n - n_j$ datos de $X_{n_{i-1}+1, n_i}$ a y parar. De lo contrario, fijar $n_{j+1} = n_j + n_i - n_{i-1} + 1$, tomar $X_{n_j+1, n_{j+1}} = X_{n_{i-1}+1, n_i}$ a, incrementar j y regresar a (*i*).

Hay quien afirma que más que una variante del esquema LZ77, el esquema LZ78 es una forma diferente de compresión basada en diccionario, ya que abandona el concepto de ventana móvil siendo su diccionario una lista potencialmente ilimitada de frases encontradas previamente en la entrada. Y es precisamente ésta mutación la que supera la deficiencia de sólo explotar los "parecidos ocurridos recientemente", ya que cualquier frase en el diccionario que se encuentre también en el mensaje fuente será codificada de manera unívoca sin desperdiciar con ésto entradas en el diccionario.

CODIFICACION LZ78 CON INNOVACION DIFERIDA

Algo que es común en los esquemas LZ77 y LZ78 es que para producir un código de salida codifican primero la cadena concordante, es decir, aquella que apareció en el diccionario, e inmediatamente después emiten sin codificar el siguiente dato en la entrada (en donde, la concatenación de la cadena concordante y el dato ya no aparecen en el diccionario) lo cual en especulaciones posteriores de otros autores [Stor92] podría ser un costo de codificación excesivo, razón por la cual dichos autores sugirieron codificar normalmente la cadena concordante (a la que ellos llamaron la cita, porque cita a una frase en el diccionario) pero diferir la codificación (porque esta vez sí se codificará) del dato (al que le llamaron la innovación, porque la concatenación de la cita con éste establece una nueva entrada en el diccionario) apareciendo como el primer dato de la siguiente cadena a codificar. Por razones obvias a esta estrategia se le llama de innovación diferida y es ejemplificada en el algoritmo descrito por Terry Welch e implementado en el programa compress del sistema operativo UNIX y su progenie.

En su artículo "A technique for High Performance Data Compression" [Wel84] de junio de 1984 Terry A. Welch describió el trabajo desarrollado en el Sperry Research Center (ahora parte de Unisys) el cual es una implementación del compresor LZ78 con innovación diferida al que él le llamo LZW.

En dicho artículo Welch también discutió el posible uso de dicho compresor en controladores de discos y cintas.

Por otra parte, el algoritmo LZ78 como originalmente fue concebido, mapea cadenas de longitud variable a códigos de longitud variable (pero predecible) y el algoritmo propuesto por Welch, mapea cadenas de longitud variable a códigos de longitud fija (normalmente de 12 bits), lo cual únicamente limita el número de entradas en el diccionario. Además, éste último algoritmo empieza con un diccionario (llamado también tabla de cadenas) que contiene todos los posibles datos atómicos en la fuente de entrada, a diferencia del LZ78, que empieza con un diccionario prácticamente vacío (conteniendo únicamente la cadena nula de longitud cero). El algoritmo propuesto por Welch es el siguiente:

- Inicializar la tabla de cadenas de modo que contenga todas las cadenas de longitud 1.
- w <- Primer dato leído de la entrada.
- Paso:
    k <- próximo dato leído de la entrada.
    Si no existe tal dato (fin de flujo de entrada):
        Salida <- código(w).
        Terminar.
    Si wk existe en la tabla de cadenas:
        w <- wk.
        Ir a Paso.
    De lo contrario (wk no está en la tabla de cadenas):
        Salida <- código(w).

Tabla de cadenas <- wk.
w <- k
ir a Paso.

Para ejemplificar el procedimiento, consideremos un alfabeto de tres caracteres (a, b y c) y la entrada ababcbababaaaaaaa, con lo que tenemos los resultados siguientes:

	a	b	a	b	c	b	a	b	a	b	a	a	a	a	a	a	a
CODIGO DE SALIDA	1	2	4	3	5		8		1	10		11					1
NUEVAS FRASES ADICIONADAS AL DICCIONARIO		4		6		8				10			12				
			5		7			9			11						

de donde la tabla de cadenas inicializada con las frases a, b y c y sus respectivos códigos (parte por arriba de la línea) queda al final del proceso como:

TABLA DE CADENAS

a	1
b	2
c	3
ab	4
ba	5
abc	6
cb	7
bab	8
baba	9
aa	10
aaa	11
aaaa	12

Debido a que la longitud de cada cadena nueva en la tabla, en principio, es impredecible y lo que es peor, puede ser lo suficientemente grande para hacer que la tabla consuma demasiado espacio en memoria, normalmente cada cadena es representada por la concatenación del código de su cadena prefijo (la cita) y su dato extensión (la innovación), con lo que para nuestro ejemplo tenemos la siguiente

TABLA DE CADENAS ALTERNATIVA

a	1
b	2
c	3
1b	4
2a	5
4c	6
3b	7
5b	8
6a	9
1a	10
10a	11
11a	12

Para decodificar los datos el descompresor utiliza la misma tabla de cadenas, y al igual que el compresor la construye conforme el proceso progresa.

Cada vez que un código es recibido el descompresor lo descifra en forma del código de una cadena prefijo (la cita) y su dato extensión (la innovación), justamente como en la tabla mostrada arriba. El dato extensión es mandado como salida y el código de la cadena prefijo es nuevamente sometido al mismo proceso. Lo anterior continúa en forma recursiva hasta que el código de la cadena prefijo es descifrado como un único dato, terminando con ésto la decodificación de un código en una cadena cuyo último dato obtenido es utilizado como el dato extensión de la cadena anterior para sumar una nueva entrada a la tabla.

Este algoritmo puede ser descrito como sigue:

```
Código <- CódigoAnterior <- Primer código de entrada.
Calcular k tal que Código = código(k)
Salida <- k
PróximoCódigo:
        CódigoActual <- Código <- próximo código de entrada
```

```
        Si no existe tal código :
            Terminar
        De lo contrario:
PróximoSímbolo:
            Si Código = código(wk):
                Salida <- k
                Código <- código(w)
                Repetir PróximoSímbolo
            De lo contrario:
                Si Código = código(k):
                    Salida <- k
                    Tabla de cadenas <- CódigoAnteriork
                    CódigoAnterior <- CódigoActual
                    Repetir PróximoCódigo
```

Realmente éste último algoritmo fue una primera aproximación dada por Welch, ya que él le encontró dos problemas. Uno de ellos, es que descifra un código en una cadena, la cual es la inversa de la original (es decir, la original tiene como primer dato el último de la descifrada, como segundo el penúltimo, etc.). Este problema puede ser resuelto utilizando una lista LIFO (o pila); de ésta forma, cada vez que un dato es obtenido del código, éste es metido a la pila y al terminar de obtener datos, éstos son sacados de la pila y mandados a la salida, con lo que ahora sí se manda la cadena original.

Para ilustrar el otro problema, consideremos la generación y emisión del código para la frase 8 en nuestro ejemplo: Cuando, al comprimir, el código para la frase 8 es mandado al diccionario, inmediatamente después dicha frase es codificada (lo cual no ocurre con la frases anteriores, por ejemplo, al ser generado el código para la frase 5, tuvieron que generarse los códigos para las frases 6 y 7 antes de que fuera emitido el código para la frase 5). Por otra parte, cuando el mensaje está siendo decodificado al llegar al código número 8, éste aún no ha sido adicionado a la tabla (nuevamente como contraparte, al llegar al código número 5 ya se han adicionado los primeros 6 códigos al diccionario) porque en la codificación se utilizó la cadena correspondiente al código anterior (es decir, la cadena correspondiente al código número 5) y el primer dato del código número 8 para calcular justamente a éste; en otras palabras, el código número 8 está definido en términos de sí mismo, por lo que para decodificarlo es necesario conocer el primer dato de la frase del cual proviene sin ayuda del diccionario.

En general, dada la secuencia kwkwk en el mensaje original, donde kw ya apareció antes en el mensaje, cuando el compresor se encuentra por primera vez la cadena kw emite el código C_k para k y adiciona el código C_{kw} para la frase kw al diccionario; y continúa el proceso hasta que es encontrada la cadena kwkwk. En este punto es emitido el código C_{kw} para kw y adicionado el código C_{kwk} para la frase kwk al diccionario, para a continuación emitir éste último al encontrarse enseguida la cadena kwk, con lo cual se presenta el fenómeno ya descrito, es decir, el código C_{kwk} para la frase kwk está definido en términos de sí mismo.

Recordando la manera en que el descompresor trabaja, dado un código, éste es descompuesto en el código de la cadena prefijo y el último dato k de la frase, repitiendo este proceso recursivamente sobre cada nuevo código obtenido de ésta forma, entonces el último dato k (al que le llamaremos el dato final) obtenido, es precisamente el primer dato de la frase decodificada.

Considerando lo anterior, Welch le dio solución a éste último problema de la forma siguiente: Al encontrarse el descompresor un código indefinido, éste puede ser descifrado sabiendo que el primer dato de la frase correspondiente a éste es el dato extensión de la cadena anterior y por tanto corresponderá al dato final de la frase actual y debido a la regularidad de la cadena kwkwk, éste también corresponderá al dato final de la cadena anterior, el cual es precisamente el último dato que se procesó.

Después de éste razonamiento Welch dió su algoritmo final.

```
CódigoAnterior <- Código <- Primer código en la entrada
Calcular k tal que Código = código(k)
Salida <- k
datofinal <- k
PróximoCódigo:
    CódigoActual <- Código <- Próximo código en la entrada
    Si no existe tal código:
        Terminar
    Si el Código no está definido (caso especial):
        Salida <- datofinal
        Código <- CódigoAnterior
        CódigoActual <- código(CódigoAnterior, datofinal)
PróximoSímbolo:
    Si Código = código(wk):
        pila <- k
        Código <- código(w)
        ir a PróximoSímbolo
    Si Código = código(k):
        Salida <- k
        datofinal <- k
        Mientras la pila no este vacía hacer:
            Salida <- pop(pila)
        Tabla de Cadenas <- CódigoAnterior, k
        CódigoAnterior <- CódigoActual
        ir a PróximoCódigo
```

Por último, si aplicamos el algoritmo a la secuencia 1, 2, 4, 3, 5, 8, 1, 10, 11, 1 de códigos obtenemos:

CODIGOS DE ENTRADA	1	2	4	3	5	8	1	10	11	1
CODIGOS DESCOMPUESTOS	a	b	1b	c	2a	5b	a	1a	10a	a
COMO CODIGO DEL PREFIJO			a		b	2a		a	1a	
Y SU DATO EXTENSION						b			a	
CADENAS DE SALIDA	a	b	ab	c	ba	bab	a	aa	aaa	a
NUEVAS FRASES										
ADICIONADAS		4		6		8		10		12
AL DICCIONARIO										
		5		7		9		11		

ESQUEMA DE CODIFICACION DE LONGITUD-CORRIDA (RLE)

Supongamos que tenemos una secuencia ordenada de dos o más tipos de símbolos. Entonces, una corrida es definida como una sucesión de uno o más símbolos idénticos, los cuales son seguidos y precedidos por un símbolo diferente o ningún símbolo. Así por ejemplo, en una taquilla de cine el arreglo de cinco hombres y cinco mujeres dado por H, M, H, M, H, M, H, M, H, M, establece diez corridas de longitud uno cada una y el arreglo H, H, H, H, H, M, M, M, M, M, establece dos corridas de longitud cinco cada una.

Es interesante hacer notar que en los anteriores ejemplos la intuición nos dice que los arreglos no son aleatorios, lo cual es implícitamente corroborado por el número de corridas y las longitudes de éstas (dos números los cuales están relacionados). En el primer arreglo las corridas son muchas y consecuentemente las longitudes de éstas disminuyen, en el segundo arreglo las corridas son pocas y tienen longitudes mayores. En otras palabras, la existencia de patrones puede ser determinada (mediante pruebas de hipótesis estadísticas) por el número de corridas y sus longitudes haciendo uso de la teoría de corridas [Gib85].

La observación anterior es porque el esquema de codificación de longitud-corrida (referido normalmente por sus siglas en inglés RLE de Run-Length Encoding) [Lyn73, Reg81] puede resultar inoperante en imágenes "razonablemente complejas", tales como imágenes escaneadas, imágenes que contienen objetos suavemente sombreados cubriendo una parte significativa de la pantalla o imágenes totalmente aleatorias. En éstos casos tal vez valdría la pena primero tomar una muestra aleatoria de pixels y realizar una prueba de hipótesis en busca de ciertos patrones y, si los hay, descartar éste esquema de codificación. Sin embargo, en la práctica muchas imágenes gráficas son apropiadas para la utilización de éste esquema, ya que muchas de sus líneas raster tienen corridas de pixels de longitud suficientemente larga. Lo que es más, a pesar de su extrema simplicidad ésta técnica es efectiva al aplicarse a la mayor parte de imágenes gráficas, por lo que es una de las más comúnmente utilizadas en la compresión de éstas.

Por otra parte, el esquema RLE trabaja sobre la base de que una secuencia de símbolos idénticos puede ser codificada como una cuenta (la cual es el número de veces que se repite el símbolo) y un identificador del símbolo repetido.

Consideremos nuevamente los dos arreglos H, M, H, M, H, M, H, M, H, M y H, H, H, H, H, M, M, M, M, M de hombres y mujeres en la taquilla de un cine, entonces de acuerdo al esquema de codificación RLE éstos serán codificados como 1H, 1M, 1H, 1M, 1H, 1M, 1H, 1M, 1H, 1M y 5H, 5M respectivamente.

Evidentemente en el primer caso más que una compresión de los datos hubo una expansión de éstos (el culpable de la cual es el patrón de corridas), lo que pone de manifiesto que existen situaciónes en las que es preferible dejar los datos sin codificar.

Debido a que un código consta de dos símbolos (la cuenta y el dato que se repite) dada una corrida de longitud menor o igual a dos no vale la pena codificarla, puesto que con ésto no obtendríamos ninguna compresión (peor aún, en un caso extremo como el anterior obtendremos una expansión de los datos). Por lo que para los ejemplos la codificación de cada uno de éstos será H, M, H, M, H, M, H, M, H, M y 5H, 5M respectivamente.

Cuando éste esquema fue ideado se utilizó el hecho de que en la codificación convencional de 8 bits de un caracter EBCDIC, cualquier caracter con un cero en la segunda posición normalmente no es empleado como dato. Por tanto tal un caracter es empleado para indicar la repetición de otros caracteres, es decir, dado el caracter $b_7 0 b_5 b_4 b_3 b_2 b_1 b_0$, significará que el siguiente caracter se repetirá $b_5 b_4 b_3 b_2 b_1 b_0 + 3$ veces, en donde si $b_5 b_4 b_3 b_2 b_1 b_0$ es cero, estamos diciendo que el caracter se repetirá 3 veces, o lo que es lo mismo, sólo codificaremos las corridas de longitud 3 o mayor, ya que como antes vimos las corridas de longitud 2 o menor no se deben codificar porque no contribuyen a la compresión de los datos.

Una variación de éste esquema es llamada salto de bloques blancos o codificación WBS (White Block Skipping). En el dominio de las imágenes gráficas sería llamado salto de bloques de fondo o BBS (Background Block Skipping). En este esquema el color de fondo es registrado y la codificación RLE es entonces llevada a cabo sobre todos los elementos que no son parte del fondo y son esos valores y cuentas de corridas los que son explícitamente salvados, junto con la localidad de inicio del segmento de corrida. Donde un segmento de corrida esté ausente, se asume que el color de fondo está implícitamente presente. Este esquema difiere de la codificación RLE en que los segmentos de corridas del fondo no son explícitamente incluidos en los datos codificados.

EL ESTANDAR JPEG DE COMPRESION DE IMAGENES DE TONOS CONTINUOS

Los programas y datos convencionales en las computadoras responden bien a la compresión basada en la explotación de variaciones estadísticas en la frecuencia de símbolos individuales o cadenas de símbolos. Desafortunadamente esos tipos de compresión no tienden a comportarse bien en imágenes de tonos continuos.

El problema principal de esos esquemas es el hecho de que los pixels en imágenes fotográficas tienden a estar bien dispersos sobre su rango entero. Si los colores en tales imágenes son graficados como un histograma de frecuencias éste no será tan "puntiagudo" como se desearía para que la compresión estadística tuviera éxito. De hecho, los histogramas para imágenes de fuentes tales como televisión tienden a ser planos. Eso significa que cada pixel tiene aproximadamente la misma probabilidad de aparición que los demás, evitando con ésto explotar las diferencias de entropía.

Los esquemas de diccionario tienen problemas similares. Las imágenes fotográficas no tienen la clase de características de los datos necesarias para crear múltiples ocurrencias de la misma frase ya que, debido a las vaguedades del mundo real, una superficie uniforme en por ejemplo, varios renglones, tenderá a ser ligeramente diferente de renglón a renglón, con lo que la concordancia de cadenas tenderá a ser pequeña limitando así la efectividad de la compresión.

En virtud de lo anterior los investigadores intentaron inicialmente aplicar las técnicas que habían funcionado en la compresión del lenguaje natural, tales como codificación diferencial y codificación adaptativa [AJ80], a la compresión de imágenes gráficas, y aunque dichas técnicas funcionaron no lo hicieron como se esperaba. La razón de ésto es que el audio es fundamentalmente diferente al video.

El audio, al estar formado por ondas senoidales tiende a ser repetitivo (con la consiguiente potencialidad de compresión), y es este tipo de patrones los que son explotados por técnicas tales como "codificación predictiva lineal" y la referida como ADPCM por sus siglas en ingles (adaptive differential pulse code modulation).

Por otra parte, a pesar de que en la década pasada hubo muchos avances en aspectos de tecnología digital alrededor de muchas aplicaciones de imágenes digitales (dispositivos especiales para adquisición de imágenes, almacenamiento de datos e impresión y despliegue de mapas de bits) la mayoría de los negocios modernos y consumidores de fotografías y otros tipos de imágenes utilizan más los medios análogos tradicionales, ya que dichas aplicaciones tienden a ser especializadas debido a su costo relativamente alto. Con la posible excepción del facsímil (fax), las imágenes digitales no

tienen un lugar común en sistemas de computación de propósito general como lo tienen el texto y las gráficas geométricas.

Nuevamente, como ya se apuntó con anterioridad en este trabajo, el obstáculo principal para muchas aplicaciones es la cantidad de datos descomunal requerida para representar una imagen digital directamente. Una versión digitalizada de una imagen a color con resolución de televisión contiene alrededor de un millón de bytes y una resolución de 35 mm requiere 10 veces esa cantidad, por lo que el uso de imágenes digitales muchas veces no es viable debido a los altos costos de almacenamiento o transmisión.

Aunque la tecnología moderna de compresión de imágenes ofrece una posible solución (las técnicas recientes del estado del arte pueden comprimir imágenes típicas de 1/10 a 1/50 de su tamaño sin comprimir sin afectación visible de la calidad de la imagen), ésta por sí sola no es suficiente, ya que hoy en día es necesario un método estándar de compresión de imágenes para habilitar la interoperabilidad de equipos de diferentes fabricantes en donde las aplicaciones de imágenes digitales que involucran almacenamiento o transmisión son muy difundidas. La recomendación CCITT del grupo 3 de máquinas fax, hoy en día presente en todas partes, es un ejemplo dramático de como un método estándar de compresión puede habilitar una aplicación de imágenes importante. Sin embargo, el método del grupo 3 trata sólo con imágenes binivel (blanco y negro) y no comprime imágenes de tonos continuos (tales como imágenes fotográficas), a color o en escala de grises (multinivel).

Videotexto, publicidad, gráficas artísticas, facsímil a color, transmisión de periódico telegráfico, imágenes médicas y muchas otras aplicaciones de imágenes de tonos continuos requieren un estándar de compresión en orden a desarrollarse significativamente más allá de su estado actual.

Y es en éste contexto que en los pocos años pasados un esfuerzo de estandarización conocido por el acrónimo JPEG (por Joint Photographic Experts Group y que fue formado por el CCITT y la ISO) ha emprendido la ambiciosa tarea de desarrollar el primer ESTANDAR INTERNACIONAL DE COMPRESION DE IMAGENES DIGITALES DE TONOS CONTINUOS (tanto a color como en escala de grises) DE PROPOSITO GENERAL [Wall91] que abarque casi todas las necesidades de las aplicaciones de imágenes de tonos continuos.

El objetivo JPEG ha sido desarrollar un método de compresión de imágenes de tonos continuos que reúna las siguientes características:

- Estar en o cerca de la compresión del estado del arte fluctuando en un amplio rango de tasas de calidad de la imagen y especialmente en el rango en el cuál la fidelidad visual a la imagen original es considerada de "muy buena" a "excelente". Dichas fluctuaciones de las tasas de calidad de la imagen y de la compresión implican un codificador parametrizable, es decir, la aplicación (o usuario) deberá poder establecer el intercambio compresión/calidad deseable.

II Ser aplicable a prácticamente cualquier clase de imagen digital de tonos continuos, es decir, no estará restringido a imágenes de ciertas dimensiones, espacios de color, tasas de aspecto de pixel (aspect ratio), etc., y no se limitará a la clase de imágenes con restricciónes en el contenido de la escena, tales como complejidad, rango de colores o propiedades estadísticas.

III Ser computacionalmente manejable para facilitar las implementaciones software con desarrollo viable sobre un rango de CPU's y las implementaciones hardware con costo viable para aplicaciones que requieren alto desarrollo.

IV Tener los siguientes modos de operación:

1) Codificación Secuencial: Cada componente (en el caso de un espacio de color RGB son 3 componentes, el Red, el Green y el Blue) de la imagen es codificado en una sola etapa (o scan) de izquierda a derecha y de arriba a abajo.

2) Codificación Progresiva: La imagen es codificada en múltiples etapas para aplicaciones en las cuáles el tiempo de transmisión es largo, y el observador prefiere ver la imagen surgir en múltiples pasos burdo-a-claro.

3) Codificación Recuperable: Este modo garantiza que la descompresión de la imagen resultará en una réplica exacta de la imagen original (aunque si bien comparado a los modos no recuperables el resultado es baja compresión).

4) Codificación Jerárquica: La imagen es codificada en múltiples resoluciones de modo que las versiones de baja resolución puedan ser accesadas sin tener primero que descomprimir la imagen en su resolución completa.

En junio de 1987 el JPEG formó 3 grupos informales de trabajo para refinar 3 métodos (los cuáles fueron seleccionados de un total de 12 métodos propuestos) basados en una tasación subjetiva de la calidad de la imagen, y en enero de 1988 un segundo proceso de selección más riguroso reveló que el "ADCT" propuesto basado en la Transformada Discreta del Coseno o DCT (Discrete Cosine Transform) aplicado a bloques de 8 X 8 produjo la imagen de mejor calidad.

En el tiempo de esa selección el método basado en la DCT fue sólo parcialmente definido para alguno de los modos de operación y de 1988 a 1990 el JPEG emprendió la tarea de definir, documentar, simular, probar, validar y simplemente ponerse de acuerdo sobre los pletóricos detalles necesarios para una genuina interoperabilidad y universalidad.

Como resultado del objetivo JPEG de ser genérico y de la diversidad de formatos de imágenes a través de las aplicaciones, el estándar propuesto contiene los 4 modos de operación ya mencionados. Para cada modo uno o más pares codificador/decodificador o codecs (encoder/decoder) distintos son especificados, aunque no es requisito que las

implementaciones deben incluir a ambos, ya que muchas aplicaciones tendrán sistemas o dispositivos los cuales requerirán sólo alguno de los dos.

En las secciones siguientes se describirá cada uno de los 4 modos de operación JPEG.

CODIFICACION SECUENCIAL BASADA EN LA TRANSFORMADA DISCRETA DEL COSENO

Las figuras siguientes ilustran los pasos del procesamiento que es el corazón de los modos de operación basados en la transformada discreta del coseno (DCT):

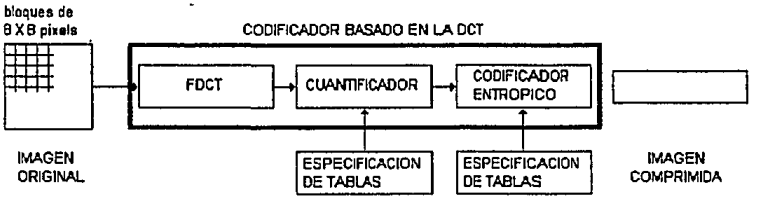


Fig. 1 Pasos del procesamiento del codificador basado en la DCT

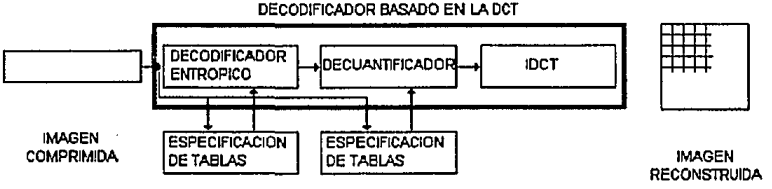


Fig. 2 Pasos del procesamiento del decodificador basado en la DCT

en donde la compresión se lleva a cabo por bloques de 8 X 8 pixels sobre un componente (escala de grises) y para imágenes a color, cada componente es comprimido completamente por separado (como si se tratara de varias imágenes en escala de grises), o los componentes son comprimidos alternadamente un bloque de 8 X 8 pixels a la vez (por ejemplo en el caso de una imagen RGB se comprime un bloque del Red, a continuación un bloque del Green y por último un bloque del Blue, etc.).

FDCT (MAPEO) E IDCT 8 X 8

En términos generales, la compresión (descompresión) basada en la DCT consiste en la aplicación de un analizador (sintetizador) armónico.

En la entrada al codificador los pixels de la imagen (significando enteros sin signo de p bits, si su profundidad es de p bits) son agrupados en bloques de 8 X 8 como lo muestra la figura 1, cambiando además el significado de las cadenas de

p bits a su interpretación en complemento a 2. Una vez hecho lo anterior cada bloque de 8 X 8 pixels se puede ver como una señal discreta de 64 puntos, la cual es una función de las coordenadas cartesianas x y y. En éste punto, el primer paso del proceso codificador consiste en la aplicación del mapeo uno a uno FDCT (Forward DCT):(señales discretas) —> (dominio de la frecuencia) a un bloque 8 X 8 de pixels, dando como resultado 64 señales fundamentales ortogonales, cada una de ellas conteniendo una de las 64 únicas "frecuencias espaciales" bidimensionales, lo que comprende el espectro de las señales de entrada. La salida del FDCT es el conjunto de las amplitudes de las 64 señales fundamentales o "coeficientes DCT", cuyos valores son univocamente determinados por la señal de entrada particular de 64 puntos.

Un análisis numérico de la ecuacion FDCT 8 X 8 revela que si las señales de entrada de 64 puntos (bloque 8 X 8) contienen pixels de p bits, entonces la parte no fraccionaria de los números de salida (coeficientes DCT) pueden crecer hasta 3 bits, lo cual paradójicamente resulta en una expansión de los datos.

Por lo anterior es importante recordar que el proceso descrito, el cual constituye la etapa de mapeo en el *proceso general* de la compresión de imágenes, es llevado a cabo para identificar las piezas de información menos relevantes y que pueden ser desechadas sin comprometer seriamente la calidad de la imagen. Debido a que normalmente los pixels varían lentamente de punto a punto a través de la imagen, la aplicación FDCT traza los fundamentos para llevar a cabo la compresión de los datos concentrando muchas de las señales en las frecuencias espaciales más bajas. En un bloque típico de 8 X 8 de una imagen típica, muchas de las frecuencias espaciales tienen amplitud cero o cercana a cero y no necesitan ser codificadas.

Los coeficientes DCT pueden de ésta forma ser vistos como las cantidades relativas de las frecuencias espaciales bidimensionales contenidas en la señal de entrada de 64 pixels. El coeficiente con frecuencia cero en ambas dimensiones es llamado el "coeficiente DC" y los 63 coeficientes restantes son llamados los "coeficientes AC". En principio el DCT no introduce pérdida en los pixels de la imagen original, sino que los transforma a un dominio en el cual pueden ser codificados más eficientemente.

Por otra parte, en el último paso del proceso decodificador (figura 2) es aplicada la función inversa de la transformada discreta del coseno (IDCT) a los 64 coeficientes DCT (los cuales en éste punto han sido cuantificados), y reconstruye la señal de salida de la imagen de 64 puntos sumando las señales fundamentales. Las ecuaciones siguientes son definiciones matemáticas idealizadas del FDCT 8 X 8 y del IDCT 8 X 8:

$$FDCT(u,v) = \frac{1}{4} C(u)C(v) \left[\sum_{x=0}^{7} \sum_{y=0}^{7} IDCT(x,y) \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right] \qquad (1)$$

$$IDCT(x,y) = \frac{1}{4} \left[\sum_{u=0}^{7} \sum_{v=0}^{7} C(u)C(v)FDCT(u,v) \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right] \qquad (2)$$

donde: $C(u), C(v) = \frac{1}{\sqrt{2}}$ para u, v = 0 y C(u), C(v) = 1 en otro caso.

Si la FDCT y la IDCT pudieran ser calculadas con precisión infinita y si los coeficientes DCT no fueran cuantificados, la señal original de 64 puntos debería ser recuperada con exactitud.

Debido a que una computadora trabaja con aritmética de precisión finita y a que la FDCT y la IDCT contienen funciones trascendentes, nunca podrán ser éstas calculadas con exactitud. Sin embargo, la importancia de las aplicaciones de la transformada discreta del coseno y su relación a la transformada discreta de Fourier ha motivado la realización de muchos algoritmos diferentes por los cuales la FDCT y la IDCT son aproximadas, y además porque un único algoritmo no es óptimo para todas las implementaciones. En este contexto, y para preservar su filosofía de libertad de innovación y costumbres en implementaciones, el JPEG ha decidido no especificar algoritmos únicos para aproximar la FDCT y la IDCT. Esto puede dar como resultado que en algunas implementaciones se den funciones bases del coseno crudamente imprecisas que podrían degradar la calidad de la imagen. Para evitar dicha degradación el standard JPEG especifica una prueba de precisión para todos los codificadores y decodificadores basados en la transformada discreta del coseno.

Por último, para cada modo de operación basado en la transformada discreta del coseno, el standard JPEG propuesto especifica codecs separados para imágenes con profundidades de 8 y 12 bits (por componente). Los codecs de 12 bits (necesarios para ciertos tipos de imágenes médicas y otras) requieren mayor poder de cómputo para alcanzar la precisión FDCT o IDCT requerida y las imágenes con otra profundidad pueden usualmente ser procesadas por codecs de 8 o 12 bits, pero eso se debe dar fuera del standard JPEG. Por lo que es responsabilidad de las aplicaciones decidir cómo ajustar o rellenar un pixel de 6 bits en la interface de entrada de codificadores de 8 bits, cómo desempaquetarlo a la salida de los decodificadores y cómo codificar cualquier información relacionada.

CUANTIFICACION

El objeto de este paso del procesamiento es descartar información la cual no es visualmente significativa. La cuantificación es un mapeo muchos a uno y por tanto es fundamentalmente no recuperable. Es la fuente principal de no recuperabilidad en los codificadores basados en la transformada discreta del coseno.

Además, es este paso el que caracteriza a los codificadores del standard JPEG como parametrizables, ya que es aquí donde la aplicación (o usuario) establecerá la relación compresión/calidad deseado por medio de una tabla de cuantificación de 64 elementos o matriz de 8 X 8 (que es justamente una transformación lineal) la cual será pasada como entrada al codificador. Entonces, el propósito de la cuantificación es obtener compresión adicional representando los coeficientes DCT en no mayor precisión de la necesaria para alcanzar la calidad de imagen deseada.

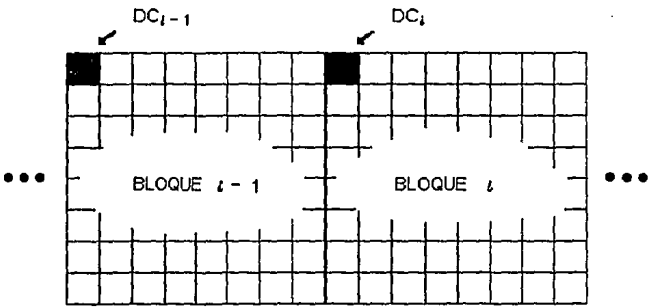
Al salir del FDCT la cuantificación de los 64 coeficientes DCT es llevada a cabo definiendo un tamaño de paso cuantificador para cada coeficiente DCT, el cual es un número entero entre 1 y 255. Hecho lo anterior, la cuantificación es llevada a cabo dividiendo cada coeficiente DCT por su correspondiente tamaño de paso cuantificador, división la cual es redondeada al entero más próximo. Entonces, si denotamos por Q a la matriz de cuantificación, la cuantificación está dada por $F^Q(u,v)$ = Redondeo al entero más próximo{F(u,v) / Q(u,v)} que no es otra cosa que normalizar cada coeficiente DCT por su correspondiente tamaño de paso cuantificador.

Obviamente la matriz de cuantificación no es suficiente para establecer los mejores umbrales cuando el objetivo es comprimir la imagen tanto como sea posible sin pérdida visible de la calidad, ya que éstos están también en función de las características de la imagen, de despliegue y de la distancia de visión. En el artículo de diciembre del '84 "A subjectively adapted image communication system" de Lohscheller H. [Loh84] es descrito un experimento que conduce a un conjunto de tablas de cuantificación para algunos tipos de imágenes y despliegues, las cuales han sido usadas experimentalmente por los miembros del JPEG y que aparecen en el standard ISO como información pero no como requerimiento.

Por otra parte, en la etapa de decuantificación del proceso decodificador es removida la normalización de cada coeficiente DCT multiplicándolo por su correspondiente tamaño de paso cuantificador, con lo que el resultado es una representación apropiada para entrada al IDCT: $(F^Q)^{-1}(u,v) = F^Q(u,v)*Q(u,v)$.

CODIFICACION ENTROPICA

El último paso del codificador es el que el JPEG llamó codificación entrópica y es llevado a cabo en 4 partes:
1) En primer lugar el coeficiente DC es tratado de manera diferente a los restantes 63 coeficientes AC en cada bloque. Debido a que usualmente existe una fuerte correlación entre los coeficientes DC de bloques 8 X 8 adyacentes, el coeficiente DC cuantificado es codificado como la diferencia del término DC del bloque previo en el orden de codificación



$\Delta DC_i = DC_i - DC_{i-1}$
Figura 3. Codificación Diferencial DC

2) En la figura anterior vemos que un bloque 8 X 8 es realmente una matriz B de 8 X 8 en donde el elemento b_{00} es el coeficiente DC y los restantes elementos de la matriz son los coeficientes AC. Y es precisamente esta matriz resultante de

la cuantificación la que nos muestra que la mayor parte de las imágenes gráficas sobre las pantallas de las computadoras están formadas de información de baja frecuencia. Todos los elementos en el renglón cero tienen un componente de frecuencia igual a cero en una dirección de la señal, todos los elementos en la columna cero tienen el otro componente de frecuencia igual a cero, y conforme nos alejamos del coeficiente DC, los demás coeficientes en la matriz representan frecuencias cada vez mayores, hallando la frecuencia más alta en b_{77}. Es decir, los coeficientes DC tienen información más relevante de la imagen (en realidad el coeficiente DC es una medida del valor promedio de los 64 coeficientes) que los coeficientes de mayor frecuencia.

También al alejarnos del coeficiente DC hallaremos que los coeficientes tienden a tener valores más pequeños y además son menos importantes para describir la imagen. Así, la transformación DCT identifica las piezas de información que pueden ser desechadas efectivamente sin comprometer seriamente la calidad de la imagen (lo cual sería extremadamente difícil de hacer en una imagen que aún no ha sido transformada).

Tomando en cuenta lo anterior el JPEG decidió ordenar los coeficientes cuantificados en una secuencia ZIG-ZAG

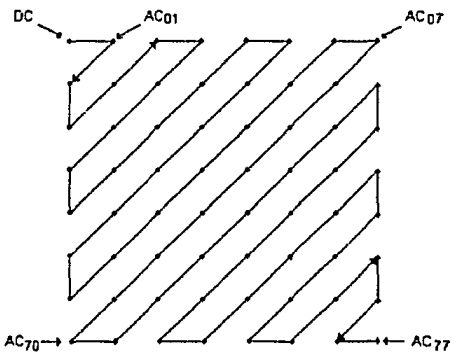


Figura 4. Secuencia ZIG-ZAG.

la cual ayuda a facilitar la codificación entrópica dejando los coeficientes de baja frecuencia (los cuales son más probables a ser distintos de cero) antes de los coeficientes de alta frecuencia.

3) Para llevar a cabo la codificación entrópica, la secuencia ZIG-ZAG de coeficientes cuantificados es primero convertida a una secuencia intermedia de pares de símbolos símbolo-1, símbolo-2, en donde símbolo-1 es a su vez un par ordenado (longitud-de-corrida, tamaño) que nos indica en su primera entrada la longitud de la corrida de coeficientes AC iguales a cero precedentes a símbolo-2, el cual es la amplitud de un coeficiente AC distinto de cero y tamaño es el número de bits utilizados para codificar símbolo-2.

La variable longitud-de-corrida puede representar corridas de ceros de longitud 0 a 15 y en el caso de corridas de ceros de longitud mayor, el valor especial (15, 0) es interpretado como un símbolo extensión de símbolo-1 con longitud de corrida igual a 16, en donde sólo puede haber 3 extensiones (15, 0) consecutivas antes de terminar símbolo-1. La terminación de símbolo-1 es siempre seguida por un solo símbolo-2, excepto para el caso en el cual la última corrida de ceros incluye el último coeficiente AC en el bloque. En este caso frecuente, el valor especial (0, 0) de símbolo-1 significa EOB (fin de bloque), y puede ser visto como un símbolo de "escape" el cual termina el bloque de 8 X 8 actual.

Debido a que la secuencia ZIG-ZAG siempre es iniciada por el coeficiente DC, o mejor dicho en éste punto, por un coeficiente DC diferencial (ya que en éste punto se han codificado las diferencias de coeficientes DC de bloques adyacentes como lo muestra la figura 3) antes de éste no existen coeficientes y por tanto, símbolo-1 sólo necesita representar la información de tamaño y no más la información de longitud-de-corrida, es decir, en éste caso símbolo-1 representa únicamente el número de bits utilizados por símbolo-2.

Por otra parte, recordando que los coeficientes DCT obtenidos de la FDCT son expandidos hasta por 3 bits, el cual es también el tamaño más grande posible de un coeficiente DCT cuantificado cuando su tamaño de paso cuantificador es igual a 1 y que el rango de valores en complemento a 2 para pixels de 8 bits de profundidad es de -2^7 a $2^7 - 1$, las amplitudes de los componentes AC cuantificados caen en el rango -2^{10} a $2^{10} - 1$. Por lo que la codificación de entero con signo usa de 1 a 10 bits para codificar a símbolo-2 (amplitud de un coeficiente AC cuantificado) y consecuentemente la componente tamaño de símbolo-1 es un número entero entre 1 y 10.

En las siguientes tablas se dá la estructura de las representaciones en símbolo-1 y símbolo-2 intermedias para los coeficientes AC cuantificados:

	ESTRUCTURA DE SIMBOLO-1		ESTRUCTURA DE SIMBOLO-2	
LONGITUD DE CORRIDA	TAMAÑO EN BITS DE LA AMPLITUD DEL COEFICIENTE AC CUANTIFICADO			AMPLITUD
	0	1 2 ... 9 10	1	-1,1
0	EOB		2	-3,-2,2,3
.	.		3	-7..-4,4..7
.	.		4	-15..-8,8..15
.	X	VALORES PERMISIBLES DISTINTOS	5	-31..-16,16..31
.	X	DE EOB Y DE ELC DE	6	-63..-32,32..63
.	X	(Longitud-de-corrida, Tamaño)	7	-127..-64,64..127
.	.		8	-255..-128,128..255
.	.		9	-511..-256,256..511
15	ELC		10	-1023..-512,512..1023

donde EOB = fin de bloque, ELC = extensión de símbolo-1 con longitud de corrida igual a 16 y X = valor no permitido

Debido a que el coeficiente DC es codificado diferencialmente éste cae en un rango 2 veces mayor del rango en el que caen los coeficientes AC, es decir, en el rango -2^{11} a $2^{11} - 1$. Así, un nivel adicional debe ser sumado al fondo de la tabla "estructura de símbolo-2" para los coeficientes DC. Con lo que, para los coeficientes DC símbolo-1 representa un valor de 1 a 11.

4) Por último es llevada a cabo la codificación entrópica propiamente dicha, la cual lleva compresión recuperable adicional codificando la secuencia de símbolos intermedios (pares símbolo-1, símbolo-2) basada en sus características estadísticas.

Tanto para los coeficientes DC como para los coeficientes AC cada símbolo-1 es codificado con un código de longitud variable (CLV) del conjunto de tablas Huffman asignado a los bloques 8 X 8 componentes de la imagen. Cada símbolo-2 es codificado con un código entero de longitud variable (ELV) cuya longitud en bits está dada en la tabla anterior.

Es importante recordar que al ser un código de longitud variable (CLV) un código Huffman, su longitud es desconocida hasta que es decodificado. No así un código entero de longitud variable (ELV), cuya longitud es almacenada en el CLV precedente.

Los códigos Huffman (CLV's) deben ser proporcionados externamente como entrada a los codificadores JPEG (figura 1) y serán representados en una forma indirecta en la imagen comprimida, forma con la cual el decodificador debe construir los mismos códigos antes de la descompresión (figura 2).

El JPEG propuesto incluye un conjunto ejemplo de tablas Huffman en su información anexa, pero debido a que son específicas de ciertas aplicaciones no son obligatorias. En contraste, los códigos ELV al ser mucho más numerosos y al estar involucrados en los CLV's pueden ser calculados en lugar ser almacenados.

Realmente, el JPEG propuesto especifica 2 métodos de codificación entrópica, la ya mencionada codificación Huffman y la codificación aritmética.

El método particular de codificación aritmética especificado en el JPEG propuesto no requiere de tablas externas de entrada, porque es capaz de adaptarse a las estadísticas de la imagen al tiempo de codificarla. La codificación aritmética ha producido de 5 a 10% mejor compresión que la codificación Huffman para muchas de las imágenes que han sido probadas por miembros del JPEG. Sí embargo, en algún sentido es más compleja que la codificación Huffman para ciertas implementaciones.

CODIFICACION RECUPERABLE PREDICTIVA

Después de la selección de un método basado en la transformada discreta del coseno en 1988, el JPEG descubrió que un modo recuperable basado también en la transformada discreta del coseno fué difícil de definir como un estándar práctico sin imponer varias restricciones sobre implementaciónes del codificador y decodificador.

Por lo que para satisfacer el requerimiento JPEG de un modo de operación recuperable se eligió un método predictivo simple el cuál es completamente independiente del procesamiento DCT descrito previamente. Aunque si bien no es el resultado de una rigurosa evaluación competitiva como lo fué el método basado en la transformada discreta del coseno, el método predictivo produce resultados los cuáles, a la luz de su simplicidad, son sorprendentemente cercanos al estado del arte para compresión recuperable de imágenes de tonos continuos.

La figura siguiente muestra los pasos del procesamiento principal para una imágen de un solo componente (escala de grises):

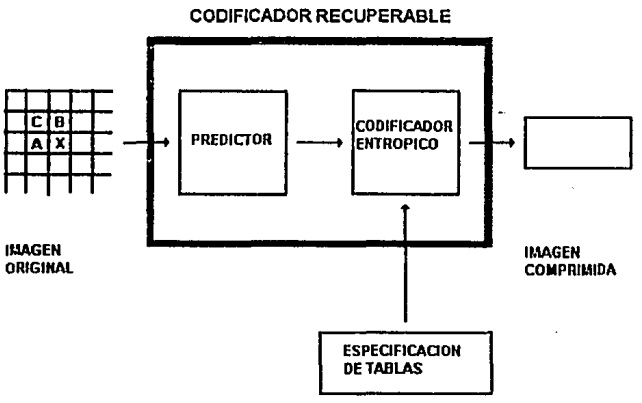


FIGURA 5. CODIFICACION RECUPERABLE PREDICTIVA

Un predictor combina los valores de hasta 3 pixels vecinos (A,B y C) para formar una predicción del pixel indicado por X, y la diferencia es codificada por alguno de los métodos Huffman o codificación aritmética. La tabla siguiente muestra los predictores (valor de selección) que pueden ser utilizados:

PREDICTORES PARA CODIFICACION RECUPERABLE

Valor de selección	predicción
0	ninguna
1	A
2	B
3	C
4	A + B - C
5	A + ((B - C) / 2)
6	B + ((A - C) / 2)
7	(A + B) / 2

Las selecciones 1,2 y 3 son predictores unidimensionales y las selecciones 4,5,6 y 7 son predictores bidimensionales. El valor de seleccion cero solo puede ser usado para codificación diferencial en el modo de operación jerárquico.

Para el modo de operación recuperable son especificados 2 diferentes codecs (uno para cada método de codificación entrópica). Los codificadores pueden utilizar cualquier profundidad de pixels de 2 a 16 bits y pueden usar cualquiera de los predictores excepto el valor de selección cero. Los decodificadores deben operar con cualquier profundidad de pixel y cualquiera de los predictores.

Los codecs recuperables producen tipicamente compresión alrededor de 2:1 para imágenes a color con escenas moderadamente complejas.

CODIFICACION PROGRESIVA DCT

El modo de operación progresivo DCT consiste del mismo FDCT y pasos de cuantificación de la sección 3.1 que son usados para el modo secuencial DCT. La diferencia clave es que cada componente de la imagen es codificado en etapas múltiples en lugar de una sola. La primer etapa codifica una versión áspera pero reconocible de la imagen, la cual puede ser transmitida rápidamente en comparación al tiempo total de transmisión, y es refinada en etapas sucesivas hasta alcanzar el nivel de calidad de imagen establecido por las tablas de cuantificación.

Para llevar a cabo lo anterior se requiere adicionalmente de un buffer de memoria del tamaño de la imagen de salida del cuantificador antes de la entrada al codificador entrópico. El buffer de memoria debe ser del tamaño suficiente para almacenar la imagen en su representación como coeficientes DCT cuantificados, cada uno de los cuales (si es directamente almacenado) es 3 bits más largo que los píxels de la imagen original. Después de que cada bloque de coeficientes DCT es cuantificado, es almacenado en el buffer de memoria para a continuación codificar parcialmente los coeficientes en cada etapa.

Existen 2 métodos complementarios por los cuales un bloque de coeficientes DCT cuantificados puede ser parcialmente codificado. Primero, solo una "banda" específica de coeficientes de la secuencia ZIG-ZAG necesita ser codificada en una etapa dada. Ese procedimiento es llamado "Selección Espectral" porque típicamente cada banda consta de coeficientes los cuales ocupan un rango de más bajo a más alto del espectro de frecuencia espacial para ese bloque de 8 X 8. Segundo, los coeficientes en la banda actual no necesitan ser codificados a su precisión total (cuantificados) en una etapa dada. En una primera codificación de los coeficientes, los N bits más significativos (donde N se debe especificar) pueden ser codificados primero. En etapas subsecuentes, los bits menos significativos pueden entonces ser codificados. Ese procedimiento es llamado "aproximación sucesiva". Ambos procedimientos pueden ser utilizados separadamente o mezclados en combinaciones flexibles.

CODIFICACION JERARQUICA

El modo jerárquico da una codificación "piramidal" de una imagen en resoluciónes múltiples, cada una diferente en resolución de su codificación adyacente por un factor de 2 en una, otra o ambas de las dimensiones horizontal o vertical. El procedimiento de codificación puede ser resumido como sigue:

a) Filtrar y reducir la muestra de la imagen original por el número deseado de múltiplos de 2 en cada dimensión.
b) Codificar esa imagen de tamaño reducido usando uno de los codificadores secuenciales DCT, progresivos DCT o recuperables descritos previamente.
c) Decodificar esa imagen de tamaño reducido, interpolar y ampliar la muestra por 2 horizontal y/o verticalmente, usando un filtro de interpolación idéntico al que usará el receptor.
d) Usar esta última imagen como una predicción de la original en esa resolución y codificar la diferencia de imágenes utilizando uno de los codificadores secuenciales DCT, progresivos DCT o recuperables descritos previamente.
e) Repetir c) y d) hasta que la imagen haya sido codificada en su resolución completa.

La codificación en los pasos b) y d) puede ser llevada a cabo utilizando sólo procesos basados en la DCT, solo procesos recuperables, o ambos.

La codificación jerárquica es útil en aplicaciones en las cuales una resolución de imagen muy alta debe ser accesada por un dispositivo de baja resolución, el cual no tiene la capacidad de buffer para reconstruir la imagen en su completa resolución y luego escalarla abajo para el despliegue de baja resolución. Un ejemplo es una imagen escaneada y comprimida en alta resolución para una impresora de muy alta calidad, donde la imagen debe ser desplegada también sobre una pantalla de video de una PC de baja resolución.

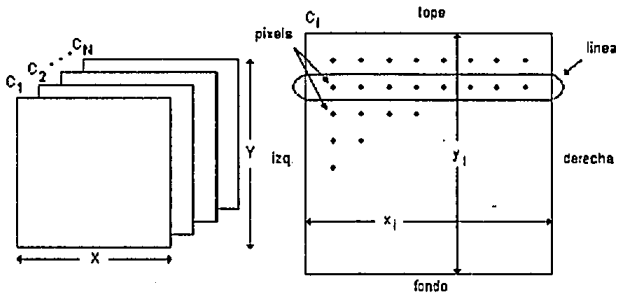
IMAGENES MULTINIVEL

En las secciones anteriores se discutieron los pasos clave del procesamiento de los codecs basados en la transformada discreta del coseno y recuperables predictivos para el caso de imágenes de un solo componente. Dichos pasos son suficientes para comprimir los datos de la imagen. Pero un buen manejo del JPEG propuesto se preocupa también por el tratamiento y control de imágenes a color u otras con múltiples componentes. El objetivo JPEG para un estándar de compresión genérico requiere que su proposición se acomode a una variedad de formatos de imágenes.

FORMATOS DE IMAGENES

El modelo de imagen utilizado en el JPEG propuesto es una abstracción de una variedad de tipos de imágenes y aplicaciones, y consiste sólo de lo que es necesario para comprimir y reconstruir los datos de la imagen digital. Debe hacerse énfasis en que el formato de datos comprimidos JPEG no codifica bastante información para servir como una representación completa de una imagen. Por ejemplo, JPEG no especifica o codifica ninguna información sobre la tasa de aspecto del pixel, espacio de color o características de adquisición de la imagen.

La siguiente figura ilustra el modelo de imagen JPEG:



a) Imágen con múltiples componentes b) Características de un componente

Figura 6. MODELO DE IMAGEN JPEG

Una imagen contiene de 1 a 255 componentes de imagen, algunas veces llamados canales o bandas espectrales o colores. Cada componente consiste de un arreglo rectangular de pixels. Un pixel es definido como un entero sin signo con p bits de precisión variando en el rango de 0 a 2^p - 1. Todos los pixels de todos los componentes en la misma imagen deben tener la misma precisión (o profundidad) p, la cual puede ser 8 o 12 para codecs basados en la DCT y 2 a 16 para codecs predictivos.

El i-ésimo componente tiene dimensiones x_i por y_i. Para acomodarse a los formatos en los cuales algunos componentes de la imagen son muestreados a diferentes tasas que otros, los componentes pueden tener dimensiones distintas. Las dimensiones deben tener una relación integral mutua definida por H_i y V_i, los factores de muestreo relativo horizontal y vertical, los cuales deben ser especificados para cada componente. Las dimensiones X, Y globales de la imagen son definidas como el máximo x_i, y_i sobre todos los componentes en la imagen, y puede ser cualquier número hasta 2^{16}. H y V sólo pueden tomar valores de 1 a 4, los parámetros codificados son X, Y y los H_i's y V_i's para cada componente. El decodificador reconstruye las dimensiónes x_i y y_i para cada componente de acuerdo a las relaciones siguientes:

$x_i = \lceil X * H_i / H_{max} \rceil$ y $y_i = \lceil Y * V_i / V_{max} \rceil$, donde $\lceil z \rceil$ es la función techo (el menor entero mayor o igual que z).

ORDEN DE CODIFICACION Y ENTREMEZCLADO

Un estándar de compresión de imágenes práctico debe preocuparse por como los sistemas necesitarán manejar los datos durante el proceso de descompresión. Muchas aplicaciones necesitarán realizar el proceso de desplegar o imprimir imágenes de múltiples componentes en paralelo con el proceso de descompresión. Para muchos sistemas lo anterior es sólo posible si los componentes son "entremezclados" en los datos comprimidos.

Para aplicar el mismo proceso de entremezclado a codecs basados en la DCT y a codecs predictivos el JPEG propuesto ha definido el concepto de "unidad de datos". Una unidad de datos es un pixel en codecs predictivos y un bloque de 8 X 8 pixels en codecs basados en la DCT.

El orden en el cual las unidades de datos comprimidos son dejadas en los datos finales comprimidos es una generalización del orden raster-scan. Generalmente, las unidades de datos son ordenadas de izquierda a derecha y de arriba a abajo de acuerdo a la orientación mostrada en la figura 6 (es responsabilidad de las aplicaciones definir cuales fronteras de una imagen son tope, fondo, izquierda y derecha). Si un componente no es entremezclado (es decir, comprimido sin ser entremezclado con otros componentes), las unidades de datos comprimidas son ordenadas en un raster-scan puro como lo muestra la figura siguiente:

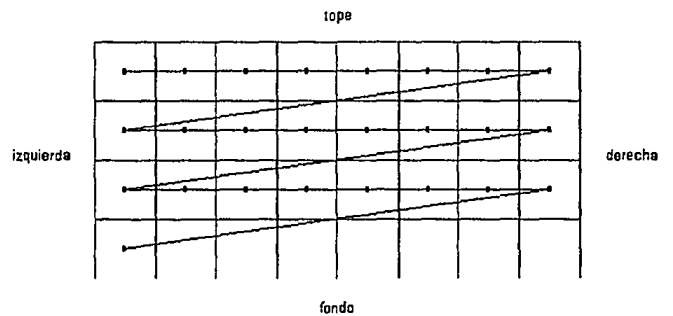
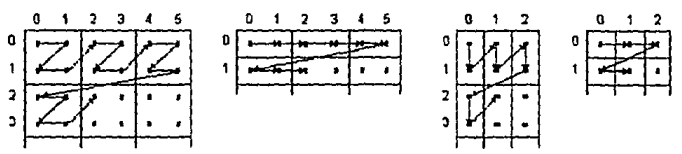


Figura 7. Ordenación de datos no entremezclados

Cuando dos o más componentes son entremezclados, cada componente C_i es particionado en regiones rectangulares de H_i por V_i unidades de datos, como muestra la figura siguiente:



$$UCM_1 = d^1_{00}, d^1_{01}, d^1_{10}, d^1_{11}, \quad d^2_{00}, d^2_{01}, \quad d^3_{00}, d^3_{10}, \quad d^4_{00},$$
$$UCM_2 = d^1_{02}, d^1_{03}, d^1_{12}, d^1_{13}, \quad d^2_{02}, d^2_{03}, \quad d^3_{01}, d^3_{11}, \quad d^4_{01},$$
$$UCM_3 = d^1_{04}, d^1_{05}, d^1_{14}, d^1_{15}, \quad d^2_{04}, d^2_{05}, \quad d^3_{02}, d^3_{12}, \quad d^4_{02},$$
$$UCM_4 = d^1_{20}, d^1_{21}, d^1_{30}, d^1_{31}, \quad d^2_{10}, d^2_{11}, \quad d^3_{20}, d^3_{30}, \quad d^4_{10},$$

Unidades de datos C_1 — C_2 — C_3 — C_4

Figura 8. Ejemplo de la ordenación de los datos en un entrelazado generalizado.

Las regiones en un componente son ordenadas de izquierda a derecha y de arriba a abajo, y en una región, las unidades de datos son ordenadas de izquierda a derecha y de arriba a abajo. El JPEG propuesto define el término "Unidad Codificada Mínima" (UCM) como el grupo más pequeño de unidades de datos entremezclados. Para el ejemplo de la figura 8, UCM_1 consiste de unidades de datos tomadas en primer lugar de la región de más a la izquierda y más arriba de C_1, seguidas por unidades de datos de la misma región de C_2, y de la misma forma para C_3 y C_4. UCM_2 continúa el mismo patrón.

Así, los datos entremezclados son una secuencia ordenada de UCM's, y el número de unidades de datos contenidas en un UCM está determinado por el número de componentes entremezclados y sus factores de muestreo relativo. el número máximo de componentes que pueden ser entremezclados es 4 y el número máximo de unidades de datos en un UCM es 10. Esa última restricción es expresada en la ecuación $\Sigma_i H_i X V_i \leq 10$, donde la suma es sobre los componentes entremezclados.

Debido a esa restricción no todas las combinaciones de los 4 componentes que pueden ser representados en orden no entremezclado en una imagen comprimida JPEG, son entremezclados. Hay que notar también que el JPEG propuesto entremezcla algunos componentes, y algunos otros no, en la misma imagen comprimida.

TABLAS MULTIPLES

Además del control de entremezclado discutido previamente, los codecs JPEG deben controlar la aplicación de la tabla de datos apropiada a los componentes apropiados. La misma tabla de cuantificación y la misma tabla de codificación entrópica (o conjunto de tablas) deben ser usadas para codificar todos los pixels en un componente.

Los decodificadores JPEG pueden almacenar hasta cuatro diferentes tablas de cuantificación y hasta cuatro diferentes (conjuntos de) tablas de codificación entrópica simultáneamente. Lo anterior es necesario para cambiar entre diferentes tablas durante la descompresión de una unidad codificada mínima, la cuál contiene múltiples componentes entremezclados, para aplicar la tabla apropiada al componente apropiado. La figura siguiente ilustra el control de cambio de tablas que debe ser manejado conjuntamente con el entremezclado de componentes múltiples por parte del codificador.

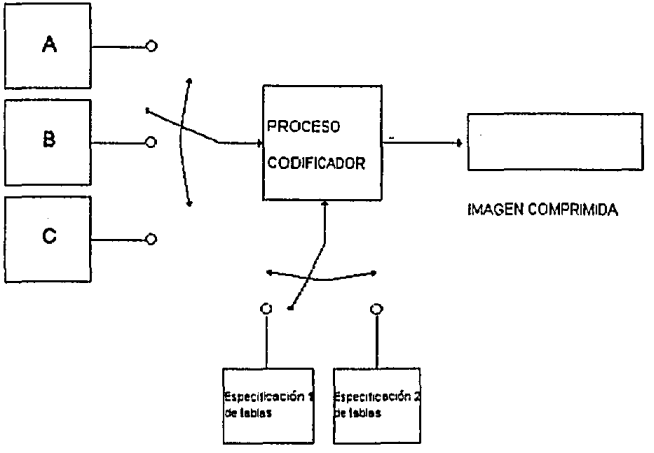


Figura 9 Control de entrelazado de componentes y cambio de tablas

Esta vista simplificada no distingue entre tablas de cuantificación y tablas de codificación entrópica.

PARTE

SEGUNDA

ESTANDARES PARA ALMACENAMIENTO DE IMAGENES

CAPITULO 4

ALMACENAMIENTO Y CODIFICACION DE IMAGENES GRAFICAS

Se ha mencionado en el capitulo anterior que el formato JPEG de datos comprimidos no codifica la información suficiente para servir como una representación completa de una imagen, ya que en una representación completa de una imagen se deben especificar otro tipo de datos además de los de la imagen misma. Tales datos incluyen desde los más obvios, como son las dimensiones de la imagen, del dispositivo de despliegue (pantalla lógica), el color de fondo de la imagen, el esquema de codificación utilizado, la versión para aplicaciónes comerciales; hasta los menos obvios como la tasa de aspecto de pixel, espacio de color (esquema donde se combinan los colores primarios, como RGB o CMYK), número de bits por pixel (profundidad del pixel), etc. [KJ92, Rim92].

Si por otra parte es adicionada dicha información a los datos de una imagen comprimida JPEG, lo que se obtendrá será una REPRESENTACION ESTATICA DE LA IMAGEN [MOP89], la cual, como se ha visto, es útil para propósitos de almacenamiento o transmisión, además de facilitar el intercambio entre sistemas heterogéneos.

Existen, de hecho, decenas de "estándares de almacenamiento" que proporcionan una representación estática para almacenamiento o intercambio de objetos gráficos. En tales estándares los OBJETOS GRAFICOS son vistos como ESTRUCTURAS DE DATOS.

Por razones obvias los formatos de esta clase se dicen ser descriptivos y orientados al contenido, y es también más o menos claro que se pueden ver como lenguajes libres de contexto, ya que una imagen cualquiera es descrita en su forma más primitiva (como un conjunto de pixels).

En contraste, existen formatos estándares que se dicen lenguajes sensibles al contexto, ya que en ésta clase una imagen es descrita en términos de PRIMITIVAS (tales como lineas, polilineas y polígonos). El objeto de estos estándares es dar una forma para crear y manipular objetos gráficos, por lo que son una REPRESENTACION DINAMICA DE UNA IMAGEN.

En ésta última clase de formatos es más obvia su naturaleza lingüística, ya que están formados de auténticos comandos que describen a la imagen (por ejemplo se puede dar un comando linea(A, B) que traza una linea con extremos A y B). También en ésta clase de formatos es posible obtener una representación estática de la imagen, pero sólo después de que el programa para generarla ha terminado de ejecutarse. De hecho, los "metafiles" gráficos han sido diseñados para éste propósito. Estos dan una vista estática o dan los medios para recrear una vista estática de los objetos gráficos representados con esta clase de estándares.

Por otra parte, la distinción entre las dos formas de representación de una imagen mencionada es natural. Para ver lo anterior considerar el problema de dibujar un rostro: A cualquiera se le ocurriría primero dibujar un óvalo (o algo parecido)

para delimitar el rostro, a continuación quizá dibujará los ojos a partir de semicírculos, y las pestañas con líneas rectas delgadas, etc. Sin embargo, al uno ver un rostro, difícilmente se imagina cómo reproducirlo a partir de las FORMAS mencionadas (aunque desde luego se puede hacer) ya que la vista tiene más la naturaleza de mapa de bits; es decir, en la medida que el ojo al percibir una imagen puede separar lo que ve en distintos objetos aislados, la imagen es un mapa de bits.

De lo anterior es que surgen las dos formas de representar una imagen: utilizando formas primitivas, por la cual se denomina una imagen vectorial (aquí el concepto de vector se refiere a cada una de las formas primitivas como son líneas, polilíneas, polígonos, círculos, etc.); y como un conjunto de puntos (pixels) a manera de malla, por la cual se denomina a una imagen de mapa de bits.

Alrededor de las dos representaciones de una imagen existen pros y contras. La principal ventaja de una imagen vectorial es que ésta puede ser transformada a placer sin degradación (un ejemplo típico son los fonts en un procesador de palabras que pueden ser cambiados de tamaño, a itálicas, etc.) y su principal desventaja es que dicha imagen no puede ser muy compleja, a diferencia de una imagen de mapa de bits, la cual como se ha visto en el capítulo anterior puede ser tan compleja como la realidad misma (al menos para el sentido visual). Pero como ya se ha repetido en múltiples ocasiones, las imágenes de mapa de bits "en el pecado llevan la penitencia" ya que entre más compleja sea una de tales imágenes, su representación directa requerirá mucho espacio de almacenamiento y poder de cómputo. Además una imagen de mapa de bits es menos flexible en su manejo, ya que si por ejemplo se desea escalar a dos veces su tamaño, se producirá un efecto conocido como aliasing o staircasing el cual consiste en que "los pixels se hacen más grandes" y producen un efecto de escaleras por toda la imagen.

Debido a que el problema que nos ocupa es el de almacenamiento de imágenes gráficas, y a que éste es característico de las imágenes de mapa de bits (también conocidas como imágenes bitmap), en las secciones siguientes se describirán algunos formatos estándares para almacenamiento de imágenes bitmap los cuales son muy utilizados para intercambio y en aplicaciones comerciales (para profundizar más en el tema se puede consultar [KJ92]).

IMG

IMG normalmente aparece como el formato de archivo propiedad de las aplicaciones GEM de Digital Research (ahora parte de Novell). El paquete GEM es una interfaz gráfica de usuario similar a Windows, excepto que carece de multitarea, manejo de memoria y otras cosas que hacen a Windows más interesante de utilizar. Quizá en compensación a esto GEM no es un procesador tan intensivo, y como regla, las aplicaciones basadas en GEM corren notablemente más rápido que aplicaciones similares corriendo bajo Windows.

En gran medida GEM no ha atraído la atención que Windows ha recibido. Hay sin embargo una excepción a eso. Una versión GEM de tiempo de corrida forma las bases de lo que se ha llegado a conocer como la versión GEM de Ventura Publisher. Ventura Publisher es casi la única aplicación basada en GEM que es comúnmente utilizada en NorteAmérica. Mientras la versión GEM de tiempo de corrida no es compatible con el GEM "real" de Digital Research, ésta ha preservado los mismos formatos de archivos bitmap y de vectores.

El formato IMG puede ser utilizado para salvar y restaurar despliegues monocromáticos, en escala de grises o de 16 colores. En modo de 16 colores, es un formato orientado a planos como PCX, y por tanto completamente eficiente con la estructura de cuatro planos de memoria EGA/VGA.

Un archivo .IMG consiste de una cabecera seguida por los datos de la imagen. Imágenes con más de un plano de color son almacenadas plano por plano desde el plano más significativo al menos significativo. Los planos son almacenados en grupos de cuatro representando los planos Red, Green, Blue y de Intensidad (RGBI).

La tabla siguiente muestra el contenido de la cabecera de un archivo .IMG.

BYTE No.	NOMBRE	DESCRIPCION
0-1	Número de versión	Esta palabra debe siempre ser fijada a 1.
2-3	Longitud en palabras de la cabecera	Una cabecera de 8 palabras es normal (Ventura Publisher crea archivos .IMG que pueden ser incompatibles con el método de despliegue descrito en la parte III de éste trabajo).
4-5	Número de planos	1 = monocromático. 2 = 16 colores
6-7	Longitud de patrón (para efectos de compresión)	De 1 a 8 bytes. Por default son 2 bytes.
8-9	Ancho del pixel	En microns (Normalmente no utilizado ya que los despliegues gráficos especifican la resolución en puntos y las impresoras en puntos por pulgada).

10-11	Altura del pixel	En microns (Normalmente no utilizado ya que los despliegues gráficos especifican la resolución en puntos y las impresoras en puntos por pulgada).
12-13	Ancho de la imagen	De hasta 65535 pixels.
14-15	Altura de la imagen	De hasta 65535 pixels.

Como se puede apreciar, la información en la cabecera es mínima. También se puede notar que no es dada información alguna de color (no se dan paletas). En modo de 16 colores, IMG utiliza los 16 colores default de la EGA, de manera que si la paleta se modificó, dichos cambios se perderán. Además, los colores son tratados de manera inusual.

Como el formato PCX, IMG utiliza una forma del esquema de longitud-corrida para comprimir los datos de la imagen. Cada plano de la imagen es almacenado como una secuencia de líneas, constando cada línea de una cuenta opcional seguida por uno o más paquetes de datos. La cuenta (si está presente) especifica el número de veces que la siguiente línea se repite en la imagen.

Una cuenta de paquetes de datos es almacenada como 00 00 FF NN (aquí un valor XY nos indica que en el nibble superior del byte está almacenada la cantidad X_{16} y en el nibble inferior la cantidad Y_{16}), en donde habrá NN copias de la siguiente línea (formada por los trozos de datos).

En una línea existen 3 tipos de paquetes, cadenas de bits, corridas de patrones, y corridas sólidas:

1) Cadenas de bits: El tipo más simple y más común representa una serie de pixels como 80 NN seguido por los NN pixels.

2) Corridas de patrones: Un grupo de pixels que se repite es almacenado como 00 NN seguido por el patrón mismo. El número de pixels en un patrón es establecido por el tamaño de patrón en la cabecera, lo que significa que todos los patrones tienen la misma longitud (normalmente 2).

3) Corridas sólidas: Una corrida sólida representa uno o más pixels ya sea todos blancos o todos negros. Un byte distinto de 00 u 80 significa una corrida sólida. Si el bit más significativo del byte está prendido, sigue una corrida de pixels negros (FF), y si dicho bit está apagado, sigue una corrida de pixels blancos (00). Los 7 bits menos significativos del byte dan la longitud de la corrida.

Para ejemplificar lo anterior supongamos que tenemos un tamaño de patrón igual a 2 y la siguiente línea:

00 00 FF 06 80 02 12 34 05 83 00 03 AB CD 80 01 88 01

Entonces, los 4 primeros bytes (00 00 FF 06) nos indican que se repetirá 6 veces la línea formada por los paquetes 80 02 12 34, 05, 83, 00 03 AB CD, 80 01 98 y 01, en donde 80 02 12 34 se traduce a 12 34, 05 se traduce a 00 00 00 00 00, 83 a FF FF FF, 00 03 AB CD a AB CD AB CD AB CD, 80 01 98 a 98 y 01 a 00, es decir, el código anterior se traduce a 6 líneas idénticas cada una conteniendo los píxels 12 34 00 00 00 00 00 FF FF FF AB CD AB CD AB CD 98 00.

Por otra parte, se podría suponer que en un archivo .IMG para 16 colores se leerá una línea de cada uno de los cuatro planos de color de la paleta EGA en turno, comprimiéndola y almacenándola, como se hace en un archivo .PCX. Pero eso no es verdad. Aunque sí bien los programas que generan archivos .IMG no pueden reconocer las paletas EGA y asumir una paleta default, lo que le hacen a los colores es totalmente distinto a lo que se podría esperar. Los colores reportados por programas generadores de archivos .IMG para la paleta estándar de 16 colores son mostrados en la siguiente tabla para dos métodos alternativos de leer los colores:

PALETA No.	DATOS IMG METODO 1	DATOS IMG METODO 2
0000	0000	0111
0001	0011	0001
0010	0101	0010
0011	0001	0011
0100	0110	0100
0101	0010	0101
0110	0100	0110
0111	1000	1111
1000	0111	0000
1001	1011	1001
1010	1101	1101
1011	1001	1011
1100	1110	1100
1101	1010	1101
1110	1100	1110
1111	1111	1000

El método 1 es la forma más directa de leer la información de color, pero eso resulta ser adecuado solo para el negro y blanco intenso. Todos los otros colores son diferentes de la paleta normal, de modo que se deberán reinicializar los registros de la paleta para que una imagen tenga la coloración apropiada. Desafortunadamente el despliegue es entonces incompatible con cualquier otra cosa en el mundo. El segundo método de leer los colores fija todos los colores adecuados excepto para los registros de paleta 0, 7, 8 y 15. Por tanto, se tendrán que cambiar esos cuatro registros para desplegar la imagen adecuadamente. Desafortunadamente, si se está utilizando el lenguaje C y se intenta utilizar printf para escribir algo en pantalla, ocurrirá que, dado que dicha función utiliza el registro 0 de la paleta como el color de fondo y el registro 7 como el color del caracter, la función producirá algo distinto que un caracter blanco sobre un fondo negro.

PCX

PCX [Zsoft91] es uno de los formatos de imágenes bitmap más viejos para PC"s y por tanto uno de los más ampliamente soportados por aplicaciones comerciales. Es el formato nativo del paquete PC Paintbrush de Z-Soft el cual en muchos aspectos fue por mucho la primera herramienta útil de dibujo disponible para usuarios de PC's.

Por otra parte, debido a que utiliza el esquema de codificación RLE para comprimir los datos de la imagen, es mejor comportado en imágenes con largas áreas de tonos constantes (por ejemplo la clase de imágenes que resultan de un paquete de dibujo) que con imágenes scaneadas o de video (lo cual ya sabíamos). Dicho esquema fué elegido buscando más bien un rápido desempaquetamiento de los datos que eficiencia. Sin embargo Z-Soft se reserva el derecho de cambiar el método de codificación para precisamente mejorar su eficiencia.

Un archivo .PCX empieza con una cabecera de 128 bytes que contiene lo siguiente:

BYTE NUMERO	TAMAÑO EN BYTES	NOMBRE	DESCRIPCION
0	1	PASSWORD (ID del fabricante)	Los archivos .PCX diseñados por Z-Soft son identificados por la contraseña 0Ah (número 10)
1	1	VERSION	Las versiones de PC Paintbrush son: 0 para la versión 2.5[1] 2 para la versión 2.8[2] con información de la paleta 3 para la versión 2.8 sin información de la paleta 4 para PC Paintbrush para Windows 5 para la versión 3.0[3] con información de la paleta
2	1	CODIGO (técnica de codificación)	Esquema de codificación utilizado 1 para codificación de longitud-corrida (RLE)
3	1	BITS POR PIXEL	Número de bits requeridos para almacenar un plano de un pixel 1 para EGA, VGA monocromo,4,8 o 16 colores o HERCULES 2 para CGA con 4 o 16 colores 4 para EGA con 16 colores 8 para VGA con 256 o 16 millones de colores
4	8	DIMENSIONES DE LA VENTANA	Coordenadas de la esquina superior izquierda (x_1, y_1) y de la esquina inferior derecha (x_2, y_2) de la ventana de despliegue ($x_i, y_i \in Z$)

[1] La versión 2.5 (la más vieja) soporta despliegues de 16 colores EGA/VGA usando la paleta de colores estándar, no son llevadas modificaciónes a la paleta.

[2] La versión 2.8 soporta despliegues de 16 colores EGA/VGA y contiene información de la paleta, que lleva a seleccionar cada uno de los 16 colores de los 64 disponibles.

[3] La versión 3.0 soporta imágenes de 256 colores EGA/VGA permitiendo definir cada uno de los 256 colores a partir de los 262,144 (2^{18}) colores posibles.

12	2	RESOLUCION HORIZONTAL	Resolución horizontal del dispositivo de despliegue (columnas) 640 para EGA Y VGA 320 para CGA o VGA con 256 colores 720 para HERCULES
14	2	RESOLUCION VERTICAL	Resolución vertical del dispositivo de despliegue (renglones) 480 para VGA con 16 colores 350 para EGA 200 para CGA o VGA con 256 colores 348 para HERCULES
16	48	MAPA DE COLORES	Información sobre la paleta de colores (inicializada a una default)
64	1	RESERVADO	Este byte no es usado actualmente por Z-Soft, por lo que cada usuario lo puede usar para identificar sus archivos, pero bajo su propio riesgo, ya que Z-Soft puede hacer uso de él en versiónes futuras
65	1	NUMERO DE PLANOS	Número de planos de color en la imagen original Normalmente 4 para EGA/VGA con 16 colores y 1 de otra forma
66	2	BYTES POR LINEA RASTER POR PLANO	Número de bytes por linea raster por plano en la imagen
68	2	DESCRIPCION DE LA INFORMACION EN LA PALETA	Cómo interpretar la paleta (usualmente ignorado) 1 para color/blanco y negro 2 para escala de grises
70	58	RELLENADOR (NO UTILIZADO)	Rellena con ceros la cabecera Ocupa hasta el fin de la cabecera.

De hecho, con excepción de la entrada 16 (Mapa de Colores) todas son autoexplicativas.

El mapa de colores o paleta default (ya que siempre es inicializado a valores default) consta de 16 conjuntos (uno para cada paleta EGA/VGA) de tres bytes cada uno (un byte para cada color primario). Debido a que en la tarjeta EGA los registros de paleta son de sólo escritura, normalmente la función que la modifica también salva la información de dichos registros en un arreglo PALETA[16] el cual es utilizado para escribir el mapa de colores. Sin embargo, en la tarjeta VGA dichos registros si se pueden leer, por lo que en éste caso existe la posibilidad de una vez desplegada la imagen leerlos y reemplazarlos por los valores default. Es interesante comentar que la limitación de sólo escritura de los registros de la EGA ha sido hábilmente superada por algunos programas de captura de pantallas: Estos "atrapan" todas las peticiónes de aplicaciones software para cambiar la paleta, indagan en la nueva paleta que está siendo fijada, y luego pasan la petición como si nunca hubiera sido interceptada.

Podemos ver como un proceso en tres etápes a la forma en que se escribe el mapa de colores en la cabecera de un archivo .PCX:

1. Se lee la i-ésima entrada del arreglo PALETA y se toman en consideración sólo los 6 bits menos significativos, los cuales contendrán información de la cantidad de RGB para la i-ésima paleta:

   Supongamos que vemos al byte leído de PALETA[i] así | | | r_i | g_i | b_i | R_i | G_i | B_i |. Entonces, las letras *mayúsculas* representan el 75% de la cantidad de un color, y las minúsculas el 25% de la cantidad del color. Por tanto, podemos tener 4 distintos estados para un color, 0%, 25%, 75% y 100% de cantidad de dicho color en la paleta correspondiente, por ejemplo; si el bit número 2 está prendido, hay al menos 75% de rojo (red), si hay más es porque el bit número 5 también está prendido y es el 100%. De lo contrario (el bit número 2 está apagado), hay 25% (el bit número 5 está prendido) o nada (el bit número 5 también está apagado) de rojo en la i-ésima paleta.

2 A continuación se combinan los dos bits correspondientes a cada color para obtener un número de cero a tres: El bit correspondiente al 25% se guarda en la posición cero del byte que le corresponde al color en cuestión en la i-ésima paleta, y el bit correspondiente al 75% se guarda en la posición 1 de dicho byte, de modo que se obtendrá cero si el color está ausente, 1 si tiene una amplitud del 25%, 2 si tiene una amplitud del 75% y 3 si tiene una amplitud del 100%.

3 El byte resultante es multiplicado por $85_{10} = 1010101_2$, de tal forma que se obtiene:

   a) $85 * 0 = 0$ si hay ausencia de color,

   b) $85 * 1 = 85 = 1010101_2$ si el color tiene una amplitud del 25%,

   c) $85 * 2 = 170 = 10101010_2$ si el color tiene una amplitud del 75% y;

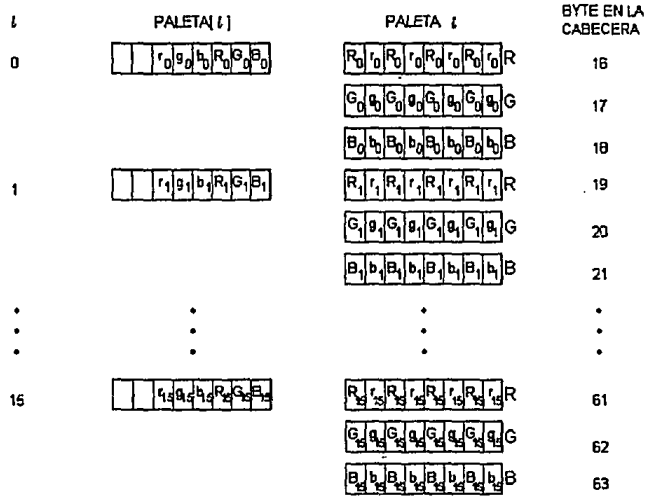
   d) $85 * 3 = 255 = 11111111_2$ si el color tiene una amplitud del 100%.

En general, para cada byte de color de cada tripleta un valor de 0 a 84 implica 0% de presencia del color, un valor de 85 a 169 implica 25% de presencia del color, un valor de 170 a 254 implica 75% de presencia del color, y un valor de 255 implica un 100% de presencia del color.

Es éste último resultado el que se escribe en el archivo .PCX

Los pasos 1 a 3 se llevan a cabo para cada $i \in \{0, \ldots, 15\}$ con lo que obtenemos el mapa de colores.

El proceso anterior se puede ver gráficamente así:



Debido a que el formato PCX es el resultado de un proceso errático de crecimiento, su diseño original no anticipo más que 16 colores, por lo que sus manifestaciones más recientes de 256 colores no pueden almacenar su mapa de colores en la cabecera y optan por pasarlo como datos finales. La forma en como es seleccionado el mapa de colores para imágenes con 256 colores es explicada en la parte 2 en el capítulo correspondiente al formato PCX.

Por otra parte, PCX es conocido también como un formato orientado a planos (en el contexto de formatos de imágenes gráficas el concepto de plano se puede considerar sinónimo del de componente). Cuando más de un plano de color es almacenado en el archivo sin comprimir (ya que en éste caso sólo manejamos imágenes con un espacio de color RGB, lo llamaremos archivo RGB) cada línea de la imagen (línea raster) es almacenada de manera ordenada (generalmente) plano por plano:

línea raster 0: RRR...GGG...BBB...
línea raster 1: RRR...GGG...BBB...
etc.

Dada esta ordenación es aplicado el esquema de codificación RLE a cada línea raster. Los detalles de la compresión serán también explicados en la parte 2 en el capítulo correspondiente al formato PCX.

TIFF

Casi todos los formatos existentes hoy en día están asociados a aplicaciónes específicas y, de hecho, muchas veces limitados al alcance del software para el cual han sido creados, siendo en éste sentido "formatos fijos" que definen una imagen sin dejar ninguna posibilidad de extensión en el estándar. Uno no puede, por ejemplo, adicionar nada a la forma en que es estructurado un archivo PCX.

Hay, sin embargo, muchas extensiónes potenciales a las imágenes básicas que pueden realzar una aplicación. Por ejemplo, sería práctico poder sumar texto a las imágenes almacenándolo como texto en lugar de "pintarlo" en la imagen.

Otra clase de extensión es la capacidad de poder desplegar una pequeña representación de una imagen mucho más grande. Dicha pequeña representación se debe poder decodificar y desplegar rápidamente por el software apropiado dando una vista previa rápida de la imagen sin tener que descomprimirla en su totalidad.

Las aplicaciónes que despliegan imágenes con una alta precisión de color o niveles de grises muchas veces requieren información que especifique la diferencia entre la forma en que las imágenes son almacenadas y la forma en que se reproducirán sobre dispositivos de salida específicos.

Las observaciones anteriores, sumadas a la necesidad de un formato de archivo gráfico que no presuponga el uso de algún software específico, condujeron a la creación del formato TIFF (Tagged image file format) [AMTIFF].

En lugar de tener una imagen "congelada" en un archivo, TIFF establece un conjunto de bloques, cada uno de los cuales comunica algo acerca de la imagen. Normalmente existe un bloque para definir la imagen misma, bloques para definir su tamaño y cómo serán manejados los colores, bloques para ayudar al software que utiliza un archivo TIFF a reproducirlo más efectivamente, bloques de texto para definir a su creador y otros detalles, etc. Cada bloque es llamado una etiqueta (tag), de ahí el nombre del formato.

La estructura de etiquetas de un archivo TIFF lo hace infinitamente extensible. Al mismo tiempo facilita a las aplicaciones que no necesiten de algunos bloques a ignorarlos. Un archivo TIFF puede contener sólo aquellos bloques que son realmente necesarios para definir la imagen, o puede contener docenas de ellos.

Entonces, el objetivo de diseño primario del formato TIFF fue proporcionar un ambiente en el cual el intercambio de imágenes entre programas de aplicación fuera posible. Dicho ambiente debería ser lo suficientemente rico para tomar ventaja de las capacidades variables de scanners y dispositivos similares. Por tanto, TIFF fue diseñado para ser un superconjunto de los formatos de archivos gráficos existentes para scanners.

Además, los diseñadores de TIFF se propusieron que éste fuera independiente de sistemas operativos específicos, sistemas de cómputo, compiladores y procesadores. La única suposición significativa es que el medio de almacenamiento

pueda soportar una secuencia de bytes de 8 bits (secuencia que es comúnmente conocida como archivo) numerados de C a N. Puesto que TIFF utiliza ampliamente el concepto de apuntador, un archivo TIFF es más fácil de leer en dispositivos de acceso aleatorio (como discos duros o flexibles), aunque es posible leer y escribir archivos TIFF sobre medios secuenciales (como cintas magnéticas).

Sin embargo, aunque en principio brillante, el formato TIFF sufre de un grado de acabado incompleto y su flexibilidad de una ligera anarquía.

En su intento por crear un formato que no se restringiría a los desarrolladores que desearan utilizarlo, los autores de TIFF dieron una especificación tan ambigua que es posible crear un archivo TIFF con la misma imagen, estructurado en muchas formas diferentes.

Las permutaciones del formato TIFF hacen casi imposible crear un lector TIFF que leerá todos los archivos TIFF utilizando una cantidad de memoria razonable.

Por otra parte, los archivos TIFF varían de acuerdo a su contenido fotométrico (color o escala de grises) y método de compresión de datos.

La especificación 5.0 define 4 clases fotométricas TIFF: TIFF-B para monocromático, TIFF-G para escala de grises, TIFF-P para colores basados en una paleta, y TIFF-R para colores RGB. Además TIFF-X es un descriptor para lectores TIFF que leen todas las clases.

Cada clase TIFF es capaz de dar un excelente desarrollo en el mantenimiento de la calidad de la imagen a través de varias plataformas y aplicaciones. Para imágenes en escala de grises, por ejemplo, TIFF permite almacenar la curva de respuesta de la imagen fuente; dicha curva permite que un lector TIFF ajuste apropiadamente la escala de grises para cualquier dispositivo de salida. Un esquema similar es empleado para imágenes a color. TIFF 5.0 permite una resolución de color de hasta 48 bits, ya sea como un color RGB o en una paleta de 64KB colores.

En todas las clases, los datos de la imagen pueden ser almacenados en cualquiera de seis formas de compresión, las cuales son referidas por un número de código:

1 Datos sin comprimir,
2 Codificación longitud-corrida Huffman modificada (CCITT grupo 3),
3 CCITT grupo 3 compatible con el Facsímile,
4 CCITT grupo 4 compatible con el Facsímile,
5 Compresión LZW y,
32773 Compresión paquetes-de-bits.

Algunas nuevas formas adicionadas en el documento 8/8/88, TIFF 5.0:

32766 dos-bits NeXT RLE,
32771 Versión de tipo 2 de palabra-alineada,
32809 cuatro-bits Thunderscan delta y RLE,
32900 Pixar "picio" RLE,
32901 Silicon Graphics RLE.

TIFF 6, liberado en los inicios de 1992 ofrece compresión JPEG y otras características nuevas.

Debido a que algunos de los esquemas de codificación mencionados son sólo variantes o mezclas de los esquemas estudiados en la parte I y con el objeto de no perder de vista la estructura de un archivo TIFF, la explicación de éstos se dará en la última sección de éste apartado.

ESTRUCTURA DE UN ARCHIVO TIFF

Un formato de archivo queda definido tanto por su forma (estructura) como por su contenido. El contenido de un archivo TIFF consiste de definiciones de campos individuales. La estructura nos dice cómo hallar dichos campos. Por varias razones no del todo obvias a primera vista la estructura es digna de serias consideraciones y, puesto que la estructura utilizada por un archivo TIFF se aparta significativamente de otros enfoques, es útil discutir la filosofía detrás de ella.

La estructura más simple y directa para un archivo de imagen es un formato posicional. En un esquema posicional, la localidad de un dato define el significado de éste. Por ejemplo, el campo para "número de renglones" podría empezar en el byte 30 (offset igual a 30).

Este enfoque es simple y fácil de implementar y es perfecto para ambientes estáticos. Pero si una cantidad significativa de cambios periódicos debe ser llevada a cabo, empiezan a presentarse problemas sutiles. Por ejemplo, supóngase que un campo debe ser reemplazado por un nuevo campo más general. Probablemente se deberá actualizar el número de versión para hacer notable el cambio. Entonces, el nuevo software no tendrá problemas con los datos de versiones anteriores, no así las versiones software anteriores, las cuales sí tendrán problemas con los datos de las versiones actualizadas. Es decir, el software anterior rechazará los nuevos datos, aunque éstos sólo se hayan modificado en un campo que tal software nunca utilizó, causando de ésta forma que el software anterior se vuelva más obsoleto de lo que probablemente se requiera.

Un enfoque para evitar ésto es almacenar una "bandera válida" para cada campo, con lo que no se tiene que actualizar el número de versión, en tanto que se pueda guardar el nuevo campo en alguna parte que no influya a ninguno de los campos anteriores. Así, el software anterior que nunca ocupó el "campo actualizado" puede continuar funcionando.

Otro problema que se manifiesta frecuentemente es que ciertos campos probablemente sólo tengan sentido si otros campos tienen ciertos valores. En la práctica sin embargo, ésto no es un problema tan serio, es sólo que hace las cosas más confusas, y también en éste sentido, la estructura de "banderas" puede ayudar a aclarar la situación.

Los programas de lectura de datos pueden ser muy útiles para propósitos de diagnóstico. Una característica deseable de tales programas es que no tengan por qué saber mucho acerca de lo que están leyendo. En particular, sería muy bueno que un programa tal leyera datos ASCII en formato ASCII, datos enteros en formato entero, etc., sin tener que, de ésta forma, "enseñarle" al programa acerca de los nuevos campos cuando éstos son adicionados. Así, puede ser que se sume un componente "tipo de dato" a nuestros campos, más información de qué tan largo es el campo. De ésta forma, el programa de lectura puede "andar" a través de los campos sin saber qué significan.

Si además se adiciona un componente "etiqueta" a cada campo, el cual nos diga qué significa el campo, podemos omitir las "banderas válidas", evitando también con ésto perder espacio en los campos no válidos en el archivo.

De acuerdo al razonamiento anterior, los creadores de TIFF le dieron a su formato una estructura de apuntadores etiquetados, los cuales indican en qué lugar del archivo (offset) están ubicados los campos de datos relacionados a una imagen específica. En donde cada conjunto de apuntadores que define una imagen es llamado un "directorio de imagen en el archivo" o IFD (Image File Directory). Además, cada uno de dichos apuntadores proporciona el tipo de datos y la longitud del campo al que apunta (capacitando así a los programas lectores a saltar los campos que no les son útiles). Este enfoque permite que los campos de datos puedan ser localizados en cualquier parte del archivo, ser aproximadamente de cualquier longitud, y contener una amplia variedad de información. Por último, TIFF ofrece la posibilidad de que haya varias imágenes relacionadas en un mismo archivo, caso en el cual habrá varios IFD's, siendo la última entrada de un IFD un apuntador al siguiente IFD en el archivo.

Por otra parte, cada etiqueta de un apuntador es un número de código que identifica la clase de datos que contiene el campo que está siendo apuntado. La especificación TIFF proporciona una lista de todos los números de etiqueta oficiales dándoles nombres útiles (por ej., el código decimal 277 corresponde al campo "número de planos"), describe qué datos identifica el apuntador, y cómo están organizados dichos datos.

Realmente la estructura hasta aquí descrita es sólo la parte intermedia de una jerarquía de tres niveles.

Un archivo TIFF empieza con una cabecera seguida por uno o más IFD's y terminando con los datos mismos. Dichos datos caen en una de 5 diferentes categorías: Básicos, Informacionales, de Facsímile, de Almacenamiento y recuperación de documentos, y "ya no recomendados" (aquellos que se hallaron poco flexibles o se hicieron obsoletos).

CABECERA

El inicio de un archivo TIFF es marcado por una cabecera de 8 bytes que apunta a uno o más IFD's:

OFFSET	LONGITUD	DESCRIPCION
0	2	Orden de los bytes: MM o II.
2	2	Número de versión: siempre 42.
4	4	Apuntador al primer IFD.

ORDEN DE LOS BYTES

La primer palabra de un archivo TIFF especifica el orden de los bytes utilizado en éste. Dicha palabra contiene la cadena ASCII "MM" o "II" ($4D4D_{16}$ o 4949_{16}). MM significa que el archivo será soportado por la arquitectura Motorola, en donde los bytes que contienen números de 16 o 32 bits son almacenados en orden de más a menos significativos. II significa que el archivo será soportado por la arquitectura Intel, en donde los bytes son almacenados en orden opuesto (de menos a más significativos).

NUMERO DE VERSION

La segunda palabra de un archivo TIFF es el "número de versión" (siempre $42 = 2A_{16}$ = '*'), el cual nada tiene que ver con la revisión corriente de la especificación TIFF. Dicho número nunca ha cambiado y probablemente nunca lo haga, ya que de ser así significará que TIFF ha cambiado en una forma tan radical que un lector TIFF tendrá problemas con la nueva versión. El número 42 fue elegido por su "profundo significado filosófico" (el cual nunca es mencionado) y puede (y debería) ser utilizado como una verificación adicional de que se está tratando con un auténtico archivo TIFF.

El que un archivo TIFF no tenga un número de versión/revisión real fue una decisión explícita de un diseño conciso. En muchos formatos de archivos, los campos de datos toman un significado diferente dependiendo de un número de versión. El problema es que al "crecer" el formato se incrementa también la dificultad para documentar qué cosas significan qué campos en una versión dada, y el software más antiguo usualmente tiende a fracasar si se encuentra con una versión más nueva. Y ya que el deseo original es que los campos TIFF tengan un significado bien definido y permanente, de modo que el software "más antiguo" pueda usualmente leer archivos TIFF "más recientes", se tomó la decisión antes mencionada.

APUNTADOR AL PRIMER IFD

Esta entrada de la cabecera contiene el offset en bytes del primer IFD. El directorio puede estar en cualquier localidad (posterior a la cabecera) del archivo pero debe empezar en un párrafo (frontera de palabra). En particular, un IFD puede seguir a los datos de la imagen que describe. Los lectores deben siempre seguir los apuntadores.

El offset en el contexto de un archivo TIFF siempre se refiere a una localidad con respecto al inicio de dicho archivo (el primer byte del archivo tiene un offset igual a cero).

DIRECTORIOS DE IMAGENES (IFD's)

Un directorio de imagen (IFD) consiste de una cuenta de 2 bytes que contiene el número de entradas (o número de campos), seguida por los apuntadores etiquetados a dichos campos de 12 bytes cada uno, y finalizando con un apuntador de 4 bytes al próximo IFD (offset del próximo IFD). Es importante hacer notar que el último IFD debe contener ceros en sus últimos 4 bytes para indicar que no hay más IFD's en el archivo. La tabla siguiente ilustra la estructura de un IFD:

OFFSET	LONGITUD	DESCRIPCION
0	2	Número de campos que definen la imagen.
2	12	Apuntador etiquetado número 0.
14	12	Apuntador etiquetado número 1.
.	.	.
.	.	.
.	.	.
n * 12 + 2	12	Apuntador etiquetado número n.
n * 12 + 4	4	Apuntador al siguiente IFD. Si no hay siguiente IFD, 0000.

NUMERO DE CAMPOS

Debido a que el número de campos que definen a una imagen puede ser variable, es útil tener presente cuántos son. Lo anterior para propósitos de saltar al se desea un IFD.

APUNTADORES ETIQUETADOS

Las entradas subsecuentes son los apuntadores a los campos que definen a la imagen o apuntadores etiquetados, los cuales son listados en orden numérico en un IFD cualquiera, una característica que ayuda a los lectores TIFF a determinar rápidamente qué campos no están presentes.

APUNTADOR AL SIGUIENTE IFD O TERMINADOR

La última entrada en un IFD son cuatro bytes de ceros, a menos que haya más de un IFD. Aunque muchas aplicaciones utilizan sólo un IFD, podría haber más para soportar características especiales, como copias múltiples de una misma imagen en diferentes resoluciones. Si hay más de un IFD, la última entrada del IFD precedente contiene un apuntador de 4 bytes al próximo IFD.

APUNTADORES ETIQUETADOS

Los apuntadores a los campos que definen la imagen tienen una estructura de 4 partes:

OFFSET	LONGITUD	DESCRIPCION
0	2	Código de la etiqueta.
2	2	Tipo de datos.
4	4	Longitud del campo.
8	4	Apuntador al campo de datos, o campo de datos mismo.

CODIGO DE LA ETIQUETA

Los primeros dos bytes de un apuntador etiquetado son el código de la etiqueta, el cual si es público, puede ser encontrado en la especificación (Aldus Corporation Developer's Desk, Microsoft Corporation Windows Marketing Group. TIFF 5.0, *An Aldus/Microsoft Technical Memorandum: 8/8/88*) [AMTIFF], la cual proporciona un listado numérico. Los códigos de etiqueta públicos son aquellos que están entre 254 y 321 (FE_{16} y 141_{16} respectivamente). Códigos de 32768 (8000_{16}) y mayores indican códigos de etiqueta privados, los cuales pueden ser asignados a compañías individuales para cubrir características propias de sus aplicaciones o características de menor interés para usuarios en general. Dichos códigos son asignados por el administrador TIFF (actualmente Aldus Developer's Desk).

TIPO DE DATOS

Los siguientes dos bytes son un código que indica el tipo de datos que contiene el campo al cual se está apuntando. TIFF soporta los siguientes tipos de datos:

1 = Entero sin signo de un byte (Tipo BYTE).
2 = Caracter de un byte (Tipo ASCII).
3 = Entero sin signo de dos bytes (Tipo SHORT).
4 = Entero sin signo de cuatro bytes (Tipo LONG)
5 = Fracción de ocho bytes (Tipo RACIONAL. Un numerador de 4 bytes seguido por un denominador de 4 bytes).

Los campos de datos ASCII deben terminar con al menos un byte 0 (Una cadena ASCII normalmente termina con el caracter nulo, en éste caso el 0). Si un tal campo termina con más de un byte 0, los demás no cuentan para determinar la longitud del campo apuntado. El propósito de los demás bytes 0 es de rellenar el párrafo (recordar que los apuntadores apuntan siempre a un byte que inicia un párrafo). Los datos ASCII almacenados en un IFD no forzosamente deben incluir un caracter nulo (byte 0).

LONGITUD DEL CAMPO

Este componente especifica el número de valores en el campo apuntado (no el número de bytes). El número de bytes utilizados por el campo apuntado puede ser calculado multiplicando la longitud del campo por el número de bytes que contiene el tipo del campo. Por ejemplo, si la longitud es de 64 y el tipo es LONG, el campo apuntado ocupa 256 bytes.

APUNTADOR AL CAMPO DE DATOS (O CAMPO DE DATOS)

Los cuatro bytes finales de un apuntador etiquetado usualmente son un apuntador al párrafo de inicio del campo de datos. Sin embargo, para salvar tiempo y espacio, si el número de bytes ocupados por dicho campo (número el cual se calcula de la forma descrita en la sección previa) es menor o igual a 4, los 4 bytes finales del apuntador etiquetado no contendrán un apuntador a un campo sino el campo mismo. Caso en el cual, el campo es justificado a la izquierda en los 4 bytes.

CAMPOS DE DATOS

Los campos de datos son los bloques de datos cuyo offset en el archivo está dado por los apuntadores etiquetados en el IFD. Los campos de datos están agrupados en 5 categorías: Básicos, Informacionales, de Facsimile, de almacenamiento y recuperación de documentos, y ya no recomendados.

Los campos básicos o informacionales son el corazón del formato TIFF. Los campos básicos definen las características de la imagen, tales como dimensiones, contenido fotométrico (color o escala de grises), y el tipo de compresión. Los campos informacionales son normalmente texto ASCII que indica el nombre del artista que creo la imagen, etc.

Los campos de facsimile y de almacenamiento y recuperación de documentos no son recomendados para su uso en intercambio con aplicaciones de publicidad Desktop, ya que caen un tanto fuera de los propósitos establecidos para el formato TIFF. Los formatos fax básicamente son implementaciones de formatos fax CCITT existentes. Los campos de almacenamiento y recuperación de documentos contienen anotaciones como el nombre de un documento asociado.

Los campos "ya no recomendados", no son recomendados excepto quizá para uso local y no deberían ser utilizados para intercambio de imágenes. Estos campos serán ya sea reemplazados por otros campos, causarán serios problemas, o simplemente no tendrán utilidad; y aún más, podrían desaparecer en una especificación futura.

CAMPOS BASICOS

Los campos básicos son los campos que son fundamentales para las características visuales de una imagen. Cada uno de los siguientes apuntadores etiquetados tiene la estructura ya descrita, en donde únicamente para propósitos de documentación son utilizados nombres de etiquetas, ya que un IFD usa códigos numéricos.

BITS POR PIXEL

Código	Tipo	Longitud	Datos
258 (102_{16})	Short	No. de planos	Profundidad del pixel.

Los Datos dan o apuntan a la profundidad del pixel .Por ej., para escala de grises, el No. de planos será 1 y por tanto la profundidad del pixel cabrá en un valor short, almacenándose este en los 4 bytes de la componente Datos. Para colores RGB sin embargo, habrá 3 planos de color, requiriéndose como consecuencia 3 pares de bytes (porque el tipo es short) con lo que el valor en la componente Datos será un apuntador a los tres valores R, G y B de 2 bytes cada uno. En éste caso el valor default es 1.

MAPA DE COLORES

Código	Tipo	Longitud	Datos
320 (140_{16})	Short	3 * 2 BITS POR PIXEL	Apuntador a 3 tablas.

Esta etiqueta apunta a 3 tablas almacenadas consecutivamente, una para cada color primario, conformando juntas una paleta de colores. Cada tabla tiene 2 BITS POR PIXEL entradas de longitud short (2 bytes). Cuando se utiliza un mapa de colores, los valores de los pixels son índices a la paleta.

COMPRESION

Código	Tipo	Longitud	Datos
259 (103_{16})	Short	1	Código entero, 1, 2, 5 o 32773.

Este campo contiene un único número de código (1, 2 o 5 cuando no es usado conjuntamente con archivos fax) que indica ya sea que los datos de la imagen están o no comprimidos, y si lo están, cómo. Un código de 1 indica datos sin comprimir; 2 indica codificación de longitud-corrida Huffman modificada uni-dimensional (Grupo 3 CCITT); 5 indica compresión LZW; 32773 indica compresión en paquetes de bits. Puesto que la especificación TIFF 5.0 fue liberada en 1988, varios esquemas de compresión han sido sumados:

32766 Es un esquema de 2-bits RLE intentado para uso con computadoras NeXT.

32771 Es una versión de palabra-alineada de compresión CCITT del tipo 2.

32809 Es un esquema de compresión para valores de 4 bits intentado para su uso con scanners Thunderscan.

32900 Es un formato RLE Pixar "picio" para imágenes de 8 y 16 bits.

32901 Es un formato RLE para Silicon Graphics similar a PackBits (paquetes de bits).

Los pixels pueden ser de cualquier longitud y si son almacenados en forma descomprimida son estrechamente empaquetados, ignorando fronteras de byte en una línea raster cualquiera. Sin embargo las líneas raster empiezan y finalizan en fronteras de byte y son rellenadas con ceros si es necesario. En el esquema de compresión CCITT del grupo 3, blanco = 0 y negro = 1 (a menos que la etiqueta de interpretación fotométrica esté presente con un valor de 1). El código de compresión default es 1.

CURVAS DE RESPUESTA DEL COLOR

Código	Tipo	Longitud	Datos
301 ($12D_{16}$)	Short	3 * 2 BITS POR PIXEL	Apuntador a 3 tablas.

Esta etiqueta da un apuntador a 3 tablas de corrección de color consecutivas, una para cada color primario. Esta es la forma en que TIFF registra el coeficiente Gamma de la imagen fuente (dicho coeficiente refleja las variaciones en la intensidad de la luz al percibir por algún medio una imagen. Por ejemplo, al capturar una imagen con un scanner "no lineal", las variaciones en la intensidad de la luz podrían estar reflejadas mediante la ecuación pixel de salida = pixel de entrada ^{Gamma}). Cada tabla tiene 2 $^{BITS\ POR\ PIXEL}$ entradas de datos short (2 bytes). Cuando una curva de respuesta del color es utilizada, los valores RGB dados por los pixels son índices a las tablas R, G y B de corrección del color. Si es utilizada una paleta, la corrección Gamma puede ser incluida en la etiqueta Mapa de Colores. El rango de valores contenidos en las tablas de corrección es de 0 a 65535, donde 0 es la intensidad mínima y 65535 es la máxima. El contenido default de éstas tablas define el estándar Gamma NTSC para monitores 2.2.

CURVA DE RESPUESTA DE GRIS

Código	Tipo	Longitud	Datos
291 (123_{16})	Short	2 $^{BITS\ POR\ PIXEL}$	Apuntador a una tabla.

Como con la etiqueta de Curva de respuesta del color, cada pixel es un índice a la tabla. Debido a que históricamente tales curvas fotométricas son registradas en un rango fraccional, los valores de 2 bytes en la tabla tienen unidades implícitas, dadas por la etiqueta unidad de respuesta de gris, desde 10^{-1} a 10^{-5} (la unidad de respuesta de gris registra el exponente sin signo). El default para éste campo es 2.

UNIDAD DE RESPUESTA DE GRIS

Código	Tipo	Longitud	Datos
290 (122_{16})	Short	1	Entero de 1 a 5.

El único entero registrado en ésta etiqueta es un valor sin signo utilizado como un exponente negativo de 10 para producir la unidad fotométrica de la Curva de respuesta de gris, es decir, 1 = 0.1, 2 = 0.01, 3 = 0.001, 4 = 0.0001, y 5 = 0.00001.

LONGITUD DE LA IMAGEN

Código	Tipo	Longitud	Datos
257 (101_{16})	Short o Long	1	No. de líneas raster.

El único valor (short o long) contenido en esta etiqueta proporciona la dimensión vertical (altura) de la imagen en términos de líneas raster.

ANCHO DE LA IMAGEN

Código	Tipo	Longitud	Datos
256 (100_{16})	Short o Long	1	No. de pixels en una línea raster.

El único valor (short o long) contenido en ésta etiqueta proporciona la dimensión horizontal (ancho) de la imagen en términos de pixels.

TIPO NUEVO SUBARCHIVO

Código	Tipo	Longitud	Datos
254 (FE_{16})	Long	1	32 banderas bits.

Cuando ésta etiqueta aparece en un IFD, indica que la imagen perteneciente a tal IFD está relacionada con una imagen en otro IFD. Sin embargo, no se dan medios para indicar cual es el IFD relacionado. La componente Datos de 4 bytes, representando (potencialmente) 32 banderas, habla de cómo las imágenes están relacionadas. Si el bit 0 esta prendido, la imagen actual es una versión de baja resolución de otra imagen. Si el bit 1 está prendido, la imagen es una página de un conjunto de imágenes (páginas múltiples). Si el bit 2 está prendido, la imagen es una máscara de 1 bit de profundidad a la que se le aplicará un AND lógico con otra imagen (pixel a pixel); Sin embargo, la etiqueta de interpretación fotométrica debe también estar presente y ser fijada a 4 para que ésta acción tome lugar. Actualmente no son utilizados los otros bits. Esta etiqueta reemplaza a la etiqueta short tipo subarchivo "no mayormente recomendada" Aunque si bien, a la fecha las banderas son idénticas.

INTERPRETACION FOTOMETRICA

Código	Tipo	Longitud	Datos
262 (106_{16})	Short	1	Código entero de 0 a 4.

El código entero contenido en ésta etiqueta determina que la imagen sea a color o monocromática, y cómo son representados los niveles de iluminación. Un código de 0 o 1 indica una imagen monocromática o en escala de grises. Cero indica valores para los pixels de 0 = blanco, $2^{\text{BITS POR PIXEL}} - 1$ = NEGRO; 1 indica lo opuesto. Un código de 2 indica colores RGB, donde 0 = la mínima intensidad y $2^{\text{BITS POR PIXEL}} - 1$ = la máxima intensidad. Un código de 3 significa que es utilizada una paleta de colores. Un código de 4 significa que la imagen es una máscara lógica (ver Tipo Nuevo Subarchivo).

CONFIGURACION PLANAR

Código	Tipo	Longitud	Datos
284 ($11C_{16}$)	Short	1	Código entero, 1 o 2.

El código entero de ésta etiqueta determina que los datos estén almacenados en un único plano (código = 1) o en planos de color (código = 2). Los pixels en TIFF son almacenados en tiras de longitud determinada por el usuario (etiqueta Cantidad de bytes en una tira), apuntadas por la etiqueta ofssets de las tiras en el IFD. Para datos RGB en un único plano, los valores aparecen en la secuencia RGBRGB ... en cada tira. Para planos de color RGB, cada tira contiene valores Red, Green o Blue; las tiras listadas en la tabla indicada los offsets de las tiras aparecen en la secuencia RGBRGB ... El default para éste campo es 1.

PREDICTOR

Código	Tipo	Longitud	Datos
317 ($13D_{16}$)	Short	1	Código entero, 1 o 0.

Esta etiqueta es utilizada por el código de compresión 5 (compresión LZW). Un código de 1 significa que no fue utilizado un esquema de predicción antes de la codificación. Cualquier otro valor significa un esquema de precompresión propuesto (pero oficialmente no válido). En éste esquema un código de N especifica un conjunto de N pixels, donde los últimos N - 1 pixels son reemplazados por la diferencia entre sus valores originales y el valor original del primer pixel. En la ausencia de ésta etiqueta se asume un código predictor de 1.

CROMATICISMO PRIMARIO

Código	Tipo	Longitud	Datos
319 ($13F_{16}$)	Racional	6	Apuntador a valores CIE.

Esta etiqueta apunta a una tabla de 6 valores que definen los colores primarios R, G y B utilizando el estándar de colorimetría CIE. Hay 3 pares de coordenadas X,Y, una para el Red, otra para el Green, y otra para el Blue. Los defaults comprenden los estándares para monitores de The Society of Motion Picture and Television Engineers (SMPTE): Red = 0.635, 0.340; Green = 0.305, 0.595; Blue = 0.155, 0.070.

UNIDAD DE RESOLUCION

Código	Tipo	Longitud	Datos
296 (128_{16})	Short	1	Código entero, de 1 a 3.

El código entero en ésta etiqueta denota la unidad de medida usada en conjunción con las etiquetas XResolución y YResolución. Un código de 1 indica que no se aplican unidades y la aplicación que lea escalará la imagen a su elección. Un código de 2 indica pulgadas, y 3 indica centímetros.

RENGLONES POR TIRA

Código	Tipo	Longitud	datos
278 (216_{16})	Short o Long	1	Valor entero.

Las imágenes TIFF están divididas en tiras de cualquier longitud dada (Cantidad de bytes en una tira); la etiqueta de renglones por tira especifica qué tantas líneas raster (renglones) son registradas por tira. En la ausencia de ésta etiqueta se asume que la imagen entera está en una sola tira (éste último enfoque no es recomendado, ya que pueden surgir problemas con el buffer o de descompresión; la especificación recomienda que las tiras sean de aproximadamente 8 KB de largo). Para máxima compatibilidad con lectores TIFF menos recientes se debe usar la forma Long, aunque los lectores más nuevos deben manejar ambos tipos.

NUMERO DE PLANOS

Código	Tipo	Longitud	Datos
277 (115_{16})	Short	1	Valor entero.

El valor contenido en ésta etiqueta determina el número de planos de color. Por ej.,1 para imágenes monocromáticas o en escala de grises, y tres para RGB. El valor default es 1.

CANTIDAD DE BYTES EN UNA TIRA

Código	Tipo	Longitud	Datos
279 (117_{16})	Short o Long	(ver abajo)	Apuntador a una tabla de longitudes de tiras

Esta etiqueta es dada como una ayuda para manejar los buffers de datos durante la descompresión. Proporciona un apuntador a una tabla que almacena la longitud comprimida de cada tira de bytes individual. Existe una entrada por tira. Si la configuración planar es 1, el número de entradas en la tabla es (longitud de la imagen + renglones por tira - 1) / renglones por tira, utilizando cálculos enteros. Si la configuración planar es 2, dados n planos en la imagen hay n veces más tiras y la tabla es n veces mayor.

OFFSETS DE LAS TIRAS

Código	Tipo	Longitud	Datos
273 (111_{16})	Short o Long	(ver abajo)	Apuntador a una tabla de offsets.

Esta etiqueta es la clave para localizar los datos bitmap. Proporciona un apuntador a una tabla que almacena los offsets de las localidades de cada tira individual. La tabla contiene una entrada por tira. El número de entradas es calculado de la misma forma que para la cantidad de bytes en una tira. Se recomienda el uso del tipo Long para *máxima* compatibilidad con lectores TIFF no tan recientes.

PUNTO BLANCO

Código	Tipo	Longitud	Datos
318 ($13E_{16}$)	Racional	2	Apuntador a coordenadas CIE.

Esta etiqueta apunta a un par de coordenadas X,Y en el diagrama de cromaticismo 1931 CIE que define el punto blanco de la imagen original [CIE86]. El default es X = 0.313; Y = 0.329, definiendo el punto blanco estándar SMPTE para monitores. El punto blanco para gráficas artísticas es diferente, y está dado por el estándar ANSI PH 2.30-1985.

XRESOLUCION

Código	Tipo	Longitud	Datos
282 ($11A_{16}$)	Racional	1	Pixels por unidad de medida de ancho.

Esta etiqueta describe la resolución de ancho original de la imagen en unidades dadas por la etiqueta unidad de resolución. Multiplicada por el ancho de la imagen proporciona el ancho físico de la imagen original.

YRESOLUCION

Código	Tipo	Longitud	Datos
283 ($11B_{16}$)	Racional	1	Pixels por unidad de longitud.

Esta etiqueta describe la resolución de longitud original de la imagen en unidades dadas por la etiqueta unidad de resolución. Multiplicada por la longitud de la imagen proporciona la longitud física de la imagen original.

CAMPOS INFORMACIONALES

Estos campos etiquetados son anotaciones a la imagen que podrían ser valiosas para el usuario. No deben ser usados para comunicar información que es esencial para reconstruir la imagen.

ARTISTA

Código	Tipo	Longitud	Datos
315 ($13B_{16}$)	ASCII	Ilimitada	Apuntador a, o nombre ASCII.

La etiqueta Artista apunta a una cadena ASCII terminada en nulo. Diseñada para ser utilizada para el nombre del autor de la imagen o para un anuncio de derechos reservados.

HORA Y FECHA

Código	Tipo	Longitud	Datos
306 (132_{16})	ASCII	20	Apuntador a una cadena ASCII que contiene hora y fecha.

Esta etiqueta apunta a una cadena ASCII terminada en nulo que contiene hora y fecha en el formato AAAA:MM:DD HH:MM:SS, utilizando un tiempo de 24 horas. La longitud de 20 caracteres incluye el espacio entre los campos y el caracter nulo de terminación.

COMPUTADORA ANFITRIONA

Código	Tipo	Longitud	Datos
316 ($13C_{16}$)	ASCII	Ilimitada	Apuntador a, o un nombre.

Cadena terminada en nulo autoexplicatoria. No constituye un substituto para el byte 0 de la cabecera.

DESCRIPCION DE IMAGEN

Código	Tipo	Longitud	Datos
270 ($10E_{16}$)	ASCII	Ilimitada	Apuntador a, o una descripción.

Identificador ASCII terminado en nulo. Por ejemplo, un nombre de persona o fotografía.

MARCA

Código	Tipo	Longitud	Datos
271 ($10F_{16}$)	ASCII	Ilimitada	Apuntador a, o nombre de un vendedor.

Cadena ASCII terminada en nulo dando el nombre de un vendedor de equipo relevante, por ejemplo, para un scanner.

MODELO

Código	Tipo	Longitud	Datos
272 (110_{16})	ASCII	Ilimitada	Apuntador a, o nombre de un modelo.

Cadena ASCII terminada en nulo dando el modelo de un equipo, generalmente la fuente de la imagen (scanner, captura de video, etc.).

SOFTWARE

Código	Tipo	Longitud	Datos
305 (131_{16})	ASCII	Ilimitada	Apuntador a, o nombre de un software.

Cadena ASCII terminada en nulo dando el nombre y versión de la aplicación.

CAMPOS FACSIMILE

Para archivos facsimile, TIFF acepta codificación CCITT grupo 3 y grupo 4. Eso es denotado por 2 datos adicionales para la etiqueta de compresión descrita previamente, mas una etiqueta de opciones grupo 3 y grupo 4.

COMPRESION

Código	Tipo	Longitud	Datos
259 (103_{16})	Short	1	Código entero, 3 o 4.

Un código 3 indica compresión CCITT grupo 3, y un código 4 indica grupo 4 [ICC85].

OPCIONES GRUPO 3

Código	Tipo	Longitud	Datos
292 (124_{16})	Long	1	32 banderas bits.

Los datos de ésta etiqueta comprenden 32 banderas bits, de los cuales sólo los 3 bits de menor orden son actualmente utilizados. Bit 0 = 0 para codificación unidimensional; 1 para codificación bidimensional. Con codificación bidimensional y múltiples tiras, cada tira debe empezar con una línea codificada unidimensionalmente. Bit 1 = 1 si los datos están descomprimidos. Bit 2 = 1 indica bits rellenados con cero para dejar el fin de línea en una frontera de byte.

OPCIONES GRUPO 4

Código	Tipo	Longitud	Datos
293 (125_{16})	Long	1	32 banderas bits.

Los datos de ésta etiqueta comprenden 32 banderas bits, de los cuales sólo los próximos al bit menos significativo (bit 1) son utilizados. Si bit 1 = 1, los datos no son comprimidos; de lo contrario, es utilizada compresión bidimensional. Cuando es utilizada compresión bidimensional, cada tira empieza en una frontera de byte, se codifica el primer renglón independientemente del precedente y se finaliza con el caracter EOFB del grupo 4.

CAMPOS DE ALMACENAMIENTO Y RECUPERACION DE DOCUMENTOS

Estos campos están disponibles por conveniencia, aunque si bien, de acuerdo a la especificación Aldus/Microsoft, no son recomendados para intercambio de imágenes entre aplicaciones de publicidad Desktop.

NOMBRE DEL DOCUMENTO

Código	Tipo	Longitud	Datos
269 ($10D_{16}$)	ASCII	Ilimitada	Apuntador o nombre.

Esta etiqueta contiene o apunta a una cadena ASCII terminada en nulo que es el nombre del documento fuente para la imagen.

NOMBRE DE PAGINA

Código	Tipo	Longitud	Datos
285 ($11D_{16}$)	ASCII	Ilimitada	Apuntador o nombre ASCII.

Esta etiqueta contiene o apunta a una cadena ASCII terminada en nulo que nombra la página en la cual la imagen fue originalmente hallada.

NUMERO DE PAGINA

Código	Tipo	Longitud	Datos
297 (129_{16})	Short	2	Número de página.

Esta etiqueta es utilizada cuando páginas múltiples de imágenes son contenidas en un archivo TIFF. El primer valor especifica el número de página asociado al IFD actual. Empezando con 0; el segundo da la cantidad total de páginas en el archivo.

XPOSICION

Código	Tipo	Longitud	Datos
286 ($11E_{16}$)	Racional	1	Apuntador a la posición X de la página.

Especifica el margen izquierdo de la imagen en la página, en unidades de resolución.

YPOSICION

Código	Tipo	Longitud	Datos
287 ($11F_{16}$)	Racional	1	Apuntador a la posición Y de la página.

Especifica el margen superior de la imagen en la página, en unidades de resolución.

ESQUEMAS DE COMPRESION

La especificación TIFF 5.0 establece muchos tipos de compresión denotados por los códigos de compresión del 2 al 5 y valores arriba de 32000. Los tipos 2, 5, 32771, 32773, 32766, 32809, 32900 y 32901 son discutidos aquí. La implementación de los tipos 3 y 4 requiere consultar el documento International Telegraph and Telephone Consultive

Committee (CCITT) Geneva, 1985. Volumen VII, fascículo VII.3, Terminal Equipment and Protocols for Telematic Services. Páginas 16 a 31 y 40 a 48, respectivamente, para facsímile grupo 3 y 4 [ICC85].

CODIGOS DE COMPRESION 2 Y 32771: CCITT GRUPO 3, CODIFICACION UNIDIMENSIONAL DE LONGITUD-CORRIDA HUFFMAN MODIFICADA

La implementación del esquema de compresión con código 2 es específica de imágenes monocromáticas, y como tal, codifica datos como corridas alternadas de pixels negros o blancos. TIFF utiliza 2 familias diferentes de códigos Huffman para corridas de pixels blancos y negros.

Una línea comprimida consta de series de códigos de longitud variable representando cada uno de ellos una corrida de pixels blancos o negros. Las corridas de blancos y negros se codifican alternadamente.

Dependiendo de la longitud de la corrida se puede necesitar más de un código para codificarla:

Para longitudes de corrida de 0 a 63, en incrementos de 1, se utilizan ***códigos terminales***.

Para longitudes de corrida de 64 a 1728, en incrementos de 64, se utilizan ***códigos compuestos***.

Para longitudes de corrida de 1729 a 2560, en incrementos de 64, se utilizan ***códigos compuestos adicionales***. En este rango, los códigos para corridas de pixels blancos y negros son los mismos.

Cualquier corrida con longitud de 0 a 2623 (2560 + 63) puede por tanto ser codificada utilizando un código compuesto seguido por un código terminal. Las corridas que excedan esa longitud utilizarán uno o más códigos "2560", seguidos por un código compuesto y un código terminal (toda corrida debe finalizar con un único código terminal).

Todas las líneas codificadas deben empezar con una corrida de blancos (la cual tendrá longitud cero si la imagen empieza con pixels negros). Para el tipo 2 de compresión, todas las líneas empezarán en una frontera de byte, y para el tipo 32771 en una frontera de palabra (2 bytes o párrafo). Los bits de relleno son utilizados al final de la línea. No existe un código especial para fin de línea. Las longitudes de las líneas son determinadas a partir de la etiqueta Ancho de la imagen.

A continuación se dá la tabla de códigos Huffman utilizada en éste esquema:

LONGITUD DE CORRIDA	CODIGO PARA CORRIDA DE BLANCOS	CODIGO PARA CORRIDA DE NEGROS
0	00110101	0000110111
1	000111	010

2	0111	11
3	1000	10
4	1011	011
5	1100	0011
6	1110	0010
7	1111	00011
8	10011	000101
9	10100	000100
10	00111	0000100
11	01000	0000101
12	001000	0000111
13	000011	00000100
14	110100	00000111
15	110101	000011000
16	101010	0000010111
17	101011	0000011000
18	0100111	0000001000
19	0001100	00001100111
20	0001000	00001101000
21	0010111	00001101100
22	0000011	00000110111
23	0000100	00000101000
24	0101000	00000010111
25	0101011	00000011000
26	0010011	000011001010
27	0100100	000011001011
28	0011000	000011001100
29	00000010	000011001101
30	00000011	000001101000
31	00011010	000001101001
32	00011011	000001101010
33	00010010	000001101011
34	00010011	000011010010
35	00010100	000011010011
36	00010101	000011010100
37	00010110	000011010101
38	00010111	000011010110
39	00101000	000011010111

40	00101001	000001101100
41	00101010	000001101101
42	00101011	000011011010
43	00101100	000011011011
44	00101101	000001010100
45	00000100	000001010101
46	00000101	000001010110
47	00001010	000001010111
48	00001011	000001100100
49	01010010	000001100101
50	01010011	000001010010
51	01010100	000001010011
52	01010101	000000100100
53	00100100	000000110111
54	00100101	000000111000
55	01011000	000000100111
56	01011001	000000101000
57	01011010	000001011000
58	01011011	000001011001
59	01001010	000000101011
60	01001011	000000101100
61	00110010	000001011010
62	00110011	000001100110
63	00110100	000001100111
64	11011	0000001111
128	10010	000011001000
192	010111	000011001001
256	0110111	000001011011
320	00110110	000000110011
384	00110111	000000110100
448	01100100	000000110101
512	01100101	0000001101100
576	01101000	0000001101101
640	01100111	0000001001010
704	011001100	0000001001011
768	011001101	0000001001100
832	011010010	0000001001101

896	011010011	0000001110010
960	011010100	0000001110011
1024	011010101	0000001110100
1088	011010110	0000001110101
1152	011010111	0000001110110
1216	011011000	0000001110111
1280	011011001	0000001010010
1344	011011010	0000001010011
1408	011011011	0000001010100
1472	010011000	0000001010101
1536	010011001	0000001011010
1600	010011010	0000001011011
1664	011000	0000001100100
1728	010011011	0000001100101
1792	00000001000	Los códigos para corridas de negros
1856	00000001100	son idénticos que para corridas de blancos
1920	00000001101	
1984	000000010010	
2048	000000010011	
2112	000000010100	
2176	000000010101	
2240	000000010110	
2304	000000010111	
2368	000000011100	
2432	000000011101	
2496	000000011110	
2560	000000011111	

CODIGO DE COMPRESION 5: COMPRESION LZW

El esquema de compresión LZW utilizado por TIFF es el mismo que el empleado por GIF cuando GIF comprime datos de 8 bits ("tamaño de código" = 8 en terminología GIF). En la siguiente sección de éste capítulo se explican los detalles de implementación.

A diferencia de GIF, la compresión LZW en TIFF opera sólo sobre datos de 8 bits. Para obtener datos de entrada de 8 bits, TIFF hace caso omiso de las fronteras de pixel, es decir, pixels de 4 bits son combinados, o pixels de 16 bits son

truncados. La compresión LZW en TIFF es aplicada de manera independiente a cada tira, sea cual fuere la configuración planar.

CODIGO DE COMPRESION 32766: NeXT RLE

Este esquema de compresión ajusta bien con los dos bits por pixel del despliegue NeXT. un valor del pixel de 0 es negro, y 3 es blanco. cualquier parte omitida en un renglón es blanca.

Los pixels son empaquetados en bytes con el pixel de más a la izquierda en los dos bits más significativos ($C0_{16}$) del byte. Un byte con un valor de 00_{16} significa una línea raster "literal". Dicha línea es seguida por los datos "literales" para la línea entera. Un byte con un valor de 40_{16} significa un paquete "literal", el cual es seguido por 4 bytes, siendo los dos primeros un offset en el renglón y los dos últimos la longitud en bytes del paquete.

En seguida de lo anterior siguen los datos de las longitudes dadas. El resto de la línea es blanco.

Cualquier otro valor indica datos RLE. Para cada byte, los dos bits más significativos dan el valor del pixel, y los 6 bits menos significativos la cuenta de repetición.

La cuenta de repetición debe fluctuar entre 1 y 63, dando el número de veces que el pixel aparecerá en el renglón. Los bytes de datos RLE deben llenar con exactitud el renglón.

CODIGO DE COMPRESION 32809: CODIFICACION THUNDERSCAN RLE Y DELTA

Este esquema codifica pixels de 4 bits con un código RLE y delta (diferencias pixel a pixel). Cada byte contiene en sus dos bits más significativos ($C0_{16}$) un tipo de código y en sus seis bits menos significativos ($3F_{16}$) el dato correspondiente.

Si el tipo de código es 3, los cuatro bits menos significativos del dato son el valor del pixel. Si el tipo de código es 2, el dato son 2 códigos delta de tres bits, con el primer código estando situado en los bits más significativos (38_{16}) del dato, y el segundo en los bits menos significativos (07_{16}). Cada código delta define un pixel en relación al anterior pixel del renglón como se muestra en las tablas dadas al final de ésta sección. Un código delta de 4 es ignorado.

Si el tipo de código es 1, el dato son tres códigos delta de dos bits, estando el primer código situado en los bits más significativos del dato (30_{16}), el segundo en los bits de enmedio ($0C_{16}$), y el tercero en los bits menos significativos (03_{16}). Un código delta de 2 es ignorado.

Por último, un tipo de código de 0 significa que el pixel previo en la línea se repetirá el número de veces dado por los seis bits menos significativos del byte. Una cuenta de repetición igual a cero es ignorada.

A continuación se dá el significado de los códigos delta utilizados en la codificación:

CODIGOS DELTA DE 3 BITS		CODIGOS DELTA DE 2 BITS	
Código	Significado	Código	Significado
0	pixel igual al anterior	0	pixel igual al anterior
1	pixel previo + 1	1	pixel previo + 1
2	pixel previo + 2	2	Saltar éste código
3	pixel previo + 3	3	pixel previo - 1.
4	Saltar éste código		
5	pixel previo - 3		
6	pixel previo - 2		
7	pixel previo - 1.		

CODIGO DE COMPRESION 32900: FORMATO PIXAR RLE

Este esquema es utilizado para comprimir imágenes con componentes de 8 o 16 bits por pixel. Cada pixel puede consistir de hasta 4 componentes, normalmente Alpha, Red, Green y Blue.

Un renglón es codificado como una serie de paquetes. Cada paquete empieza con un descriptor de paquete, el cual es un valor de 2 bytes (short). Los 4 bits más significativos del short indican el tipo de paquete, y los 12 bits menos significativos + 1 son una cuenta de repetición, es decir, si dichos 12 bits almacenan un 0, la cuenta de repetición es 1. Un paquete del tipo 1 es un paquete de "vaciado completo". La cuenta de repetición en éste caso indica el número de pixels que siguen al descriptor.

Un paquete del tipo 2 es un paquete de "corrida completa". La cuenta de repetición especifica el número de pares (cuenta_rep, pixel). En cada par, la cuenta de repetición es un byte o short del tamaño de un componente, y el pixel es repetido cuenta_rep + 1 veces.

Un paquete del tipo 3 es un paquete de "vaciado parcial". El descriptor es seguido por un valor byte o short (dependiendo del tamaño del componente), el cual es utilizado como el primer componente (constante) de todos los pixels descritos por éste paquete (normalmente el componente Alpha en imágenes con 4 componentes).

Siguiendo al componente constante se encuentran los valores de los componentes restantes de los pixels (es decir, siguen los pixels con un componente menos que los utilizados por la imagen). Notar que éste tipo sólo tiene sentido en una imagen de múltiples componentes que es almacenada en un único en lugar de varios planos de componente separados.

Un paquete del tipo 4 es un paquete de "corrida parcial". El descriptor es seguido por una componente constante. Todos los pixels descritos por éste paquete utilizan la constante como su primer componente. Siguiendo al valor constante hay pares (cuenta_rep, pixel). En cada par, la cuenta de repetición es un byte o short del tamaño del componente, y el pixel es repetido cuenta_rep + 1 veces. Como en un paquete de "vaciado parcial", los pixels descritos tienen un componente menos de los que utiliza la imagen.

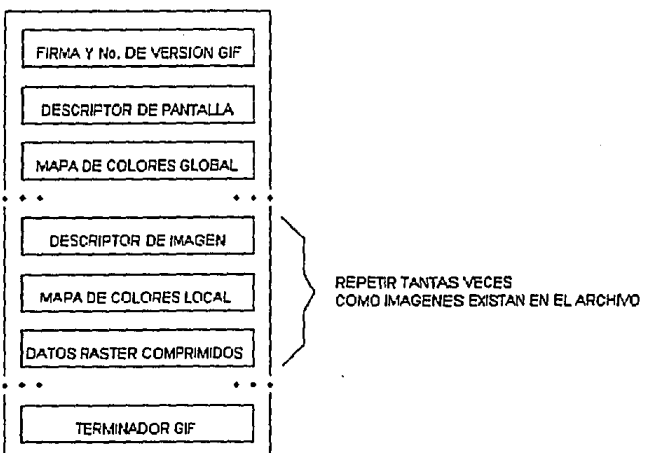
GIF

El formato GIF [Com90] desarrollado por Compuserve para transmisión en línea de imágenes a color a través de su red fue diseñado con varias características específicas en mente. A diferencia de por ejemplo el formato PCX las imágenes GIF utilizan números de color (de acuerdo al modo de trabajo de muchas targetas VGA) en lugar de colores directos RGB. Su esquema de compresión fue elegido buscando la máxima compresión posible de la imagen en lugar de rápido desempaquetamiento. Contiene suficiente información organizada lo bastante bien para que un amplio rango de diferentes dispositivos de entrada y salida puedan intercambiar imágenes.

El estándar GIF actual posibilita la creación de archivos con algunas de las extensiones y opciones ofrecidas por el formato TIFF, pero sin las ambigüedades e incompatibilidades por las cuales los archivos TIFF son legendarios. Excepto por limitaciones de espacio de almacenamiento o hardware de despliegue (o archivos corruptos) virtualmente todos los archivos GIF creados apropiadamente pueden ser leídos por cualquier lector GIF.

Además de su imagen primaria un archivo GIF puede contener BLOQUES DE EXTENSION los cuales lo capacitan para guardar imágenes múltiples (que pueden ser usadas como logos o algún otro dato que se desee aparezca sobre la imagen primaria y luego se desvanezca). Los bloques de extensión pueden también definir cómo serán tratadas las múltiples imágenes. Estos también pueden contener texto (como mensajes de derechos reservados, etc.) o información específica de la aplicación, capacitando a las aplicaciones especializadas a extender el formato para su uso propio.

Un archivo .GIF está definido en términos de bloques y sub-bloques (donde un bloque es una colección de bytes) los cuales contienen parámetros y datos relevantes utilizados en la reproducción de una imagen. Cada bloque es identificado por una etiqueta en su primer byte. El formato general de un archivo .GIF se puede ver gráficamente así:



en donde:

- El bloque de cabecera (que contiene la firma y número de versión GIF), identifica el archivo como un archivo GIF válido e indica la versión del decodificador (87a u 89a) que se requiere para interpretar adecuadamente los datos siguientes.

- El bloque descriptor de pantalla lógica, define el tamaño, tasa de aspecto de pixel, y profundidad de un plano para las imágenes en el archivo, además de indicar si a continuación hay un mapa global de colores.

- El mapa de colores global (paleta global), si se especificó, constituye una paleta default de tripletas RGB de 24 bits utilizada en imágenes subsecuentes que no cuenten con un mapa de colores local.

- Uno o varios conjuntos de tres bloques (el descriptor de imagen, mapa de colores local opcional, y los datos de la imagen comprimidos), contiene la imagen o imágenes en el archivo.

  El descriptor de imagen define las dimensiones y paleta de la imagen, y su posición en la pantalla lógica. El mapa de colores local (si se especificó uno) establece la paleta para ésta (y sólo para ésta) imagen. Los datos de la imagen comprimidos de acuerdo al algoritmo LZW son divididos en sub-bloques contiguos de hasta 256 bytes.

  Es importante resaltar que éste formato corresponde a la versión 87a, ya que la versión 89a adiciona bloques de extensión que son bloques de propósito especial y que pueden contener un código de aplicación especial o comentarios no imprimibles. Además, cuando se desea una secuencia especial de imágenes (como por ejemplo una animación), las imágenes son precedidas por un "bloque de interpretación gráfica" (el cual es un bloque de extensión) que detalla el proceso.

- El bloque terminador GIF le pone fin a los datos en el archivo.

En las secciónes siguientes se explicará cada una de éstas partes con más detalle.

CABECERA (FIRMA Y VERSION GIF)

La cabecera identifica un archivo GIF en contexto, esto es, identifica el conjunto de capacidades mínimas requeridas por un decodificador para procesar COMPLETAMENTE los datos. Obviamente, entre más baja sea la versión, dicho conjunto de capacidades también será menor; es por ésto que se recomienda que los codificadores usen la versión más baja posible que abarque todos los bloques utilizados para el archivo, de tal forma que un mayor número de decodificadores puedan procesar con éxito ta(s) imagen(es). Sin embargo, si un decodificador encuentra un número de versión el cual no es capaz de procesar completamente, aún así debe intentar procesar los datos dando su máxima capacidad, después de lo cual mandará quizá un mensaje al usuario de que los datos están incompletos.

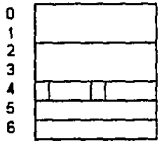
La cabecera GIF consta de 6 bytes. En los tres primeros se guarda la firma "GIF" que para propósitos de transmisión sirve como una indicación de que el flujo que llega es una secuencia de bloques GIF. En los tres últimos bytes de la

cabecera es guardado el número de versión (hasta ahora "87a" u "89a"), ordenado en forma creciente en sus dos primeros dígitos empezando en 87 (87, 88, ..., 99, 00, ..., 85, 86), y alfabeticamente en su tercer caracter (a, ..., z).

DESCRIPTOR DE PANTALLA LOGICA

El descriptor de pantalla lógica contiene los parámetros necesarios para definir el área del dispositivo de despliegue on la cual las imágenes serán vaciadas. Las coordenadas en éste bloque son dadas con respecto a la esquina superior izquierda de la pantalla virtual, que no necesariamente se refieren a las coordenadas absolutas del dispositivo de despliegue. Eso implica que referirán coordenadas en una ventana en un ambiente basado en ventanas o coordenadas de impresora cuando una impresora es utilizada. Este bloque debe aparecer inmediatamente después de la cabecera.

La sintaxis de éste bloque es:



CAMPO	TIPO
Ancho de la pantalla lógica	UNSIGNED
Alto de la pantalla lógica	UNSIGNED
<Campos empaquetados>	Ver abajo
Indice de color de fondo	BYTE
Tasa de aspecto de pixel	BYTE

<Campos empaquetados> =		
	Bandera de tabla global de colores	1 BIT (bit 7)
	Resolución de color	3 BITS (6, 5 y 4)
	Bandera sort	1 BIT (3)
	Tamaño de la tabla de colores global	3 BITS (2, 1 y 0)

donde:

- Ancho de la pantalla lógica: Ancho en pixels de la pantalla lógica donde las imágenes serán vaciadas en el dispositivo de despliegue.
- Alto de la pantalla lógica: Altura en pixels de la pantalla lógica donde las imágenes serán vaciadas en el dispositivo de despliegue.
- Bandera de tabla global de colores: Bandera que indica la presencia de una tabla de colores global. Si está prendida, inmediatamente después del descriptor de pantalla lógica seguirá la tabla global de colores, además de establecer el color de fondo como el color apuntado en dicha tabla por el índice de color de fondo (siguiente byte). Si está apagada, no seguirá una tabla global de colores y el índice de color de fondo se ignorará.
- Resolución de color: Número de bits por color primario menos uno en la imagen original. Representa el tamaño de la paleta de la cual los colores en la imagen fueron seleccionados; por ejemplo, si el valor en éste campo es 3, la paleta de la imagen original tiene 4 bits por color primario. Este valor indica la riqueza de la paleta original.

- Bandera sort: Si éste bit está prendido, la tabla global de colores está ordenada en orden de importancia decreciente (normalmente en orden de frecuencia decreciente), lo cual ayuda a un decodificador con menos colores disponibles a elegir el mejor subconjunto de colores. Si está apagada, la tabla global de colores no está ordenada.
- Tamaño de la tabla de colores global: Si la bandera de tabla global de colores está prendida, $2^{(\text{valor de éste campo} + 1)}$ nos dará el tamaño (en bytes) de la tabla de colores global. Si la bandera de tabla global de colores está apagada (es decir, no se especificó una tabla global de colores), la fórmula de arriba puede ser utilizada por los decodificadores para elegir el mejor modo gráfico para desplegar la(s) imagen(es).
- Indice de color de fondo: Indice en la tabla global de colores que apunta al color usado para aquellos pixels en la pantalla que no son cubiertos por una imagen. Si la bandera de tabla global de colores está apagada (no se especificó una tabla global de colores), éste campo debe ser cero e ignorado.
- Tasa de aspecto de pixel: Factor usado para calcular una aproximación de la tasa de aspecto de pixel en la imagen original, tasa la cual es definida como el cociente (ancho del pixel) / (altura del pixel). El rango de valores en éste campo va desde el pixel más ancho de 4:1, hasta el pixel más delgado de 1:4, con incrementos de 1/64. Si el valor en éste campo no es cero, la aproximación se calcula según la fórmula tasa de aspecto = (tasa de aspecto de pixel + 15) / 64 (donde tasa de aspecto de pixel puede ser un número entero entre 1 y 255). Si el valor en éste campo es cero, no es proporcionada información sobre el aspecto del pixel.

MAPA DE COLORES GLOBAL

Este bloque contiene una tabla (paleta) de colores, la cual es una secuencia de tripletas de bytes representando tripletas RGB de colores:

	7 6 5 4 3 2 1 0	CAMPO	TIPO
0		Red 0	BYTE
1		Green 0	BYTE
2		Blue 0	BYTE
3		Red 1	BYTE
	... ...	...	...
764		Blue 254	BYTE
765		Red 255	BYTE
766		Green 255	BYTE
767		Blue 255	BYTE

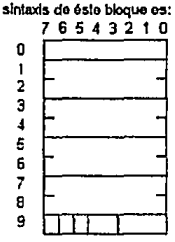
Es usada por imágenes sin una tabla de color local y por bloques de extensión de texto. Su presencia es indicada por la bandera (prendida) de tabla global de colores en el bloque descriptor de pantalla lógica, sigue inmediatamente a dicho bloque (si está presente) y contiene un número de bytes igual a $3 * 2^{(\text{tamaño de la tabla de colores global} + 1)}$.

DESCRIPTOR DE IMAGEN

Cada imagen en el archivo está compuesta de un descriptor de imagen, una tabla de colores local opcional, y los datos de la imagen. También cada imagen debe ajustarse a las fronteras de la pantalla lógica definida en el bloque descriptor de pantalla lógica.

El descriptor de imagen contiene los parámetros necesarios para procesar una imagen. Las coordenadas dadas en éste bloque refieren coordenadas en la pantalla lógica y son dadas en pixels. Este bloque es un bloque de interpretación gráfica opcionalmente precedido por uno o más bloques de control tal como el bloque de extensión de control gráfico, y puede ser opcionalmente seguido por una tabla de colores local. El descriptor de imagen es siempre seguido por los datos de la imagen.

La sintaxis de éste bloque es:



CAMPO	TIPO
Separador de imagen	BYTE
Posición izquierda de la imagen	UNSIGNED
Posición tope de la imagen	UNSIGNED
Ancho de la imagen	UNSIGNED
Altura de la imagen	UNSIGNED
<Campos empaquetados>	Ver abajo

<Campos empaquetados> =		
	Bandera de tabla de colores local	1 BIT (bit 7)
	Bandera Interlace	1 BIT (6)
	Bandera sort	1 BIT (5)
	Reservado	2 BITS (4 y 3)
	Tamaño de la tabla de colores local	3 BITS (2, 1 y 0)

donde:

- Separador de imagen: Identifica el inicio de un descriptor de imagen. Este campo contiene el valor constante 0x2C (caracter ",").

- Posición izquierda de la imagen: Número en pixels de la columna del borde izquierdo de la imagen con respecto al borde izquierdo de la pantalla lógica (La columna más a la izquierda de la pantalla lógica es la 0).

- Posición tope de la imagen: Número en pixels del renglón del borde superior de la imagen con respecto al borde superior de la pantalla lógica (El renglón superior de la pantalla lógica es 0).

- Ancho de la imagen: Ancho en pixels de la imagen.

- Altura de la imagen: Altura en pixels de la imagen.

- Bandera de tabla de colores local: Si éste bit está prendido, inmediatamente después de éste descriptor de imagen sigue una tabla de colores local. Si está apagado, no sigue una tabla de colores local y se utiliza la tabla de colores global si se hubo especificado tal.

- Bandera Interlace: Si ésta bandera está apagada, la imagen es desplegada con el interlacing normal del dispositivo de despliegue. Si está prendida, el interlacing es llevado a cabo en cuatro pasos:

Renglon de la imagen	Paso 1	Paso 2	Paso 3	Paso 4	Resultado
0	"1a"				"1a"
1				"4a"	"4a"
2			"3a"		"3a"
3				"4b"	"4b"
4		"2a"			"2a"
5				"4c"	"4c"
6			"3b"		"3b"
7				"4d"	"4d"
8	"1b"				"1b"
9				"4e"	"4e"
10			"3c"		"3c"
11				"4f"	"4f"
12		"2b"			"2b"

. . .

El primer paso despliega un renglón de la imagen cada ocho renglones, empezando en el renglón superior de la ventana de la imagen. El segundo paso despliega cada ocho renglones, empezando en el quinto renglón. El tercer paso despliega cada cuatro renglones, empezando en el tercer renglón. El cuarto paso despliega cada dos renglones, empezando en el segundo renglón.

- Bandera sort: Si éste bit está prendido, la tabla de colores local está ordenada en orden de importancia decreciente (normalmente en orden de frecuencia decreciente), lo cual ayuda a un decodificador con menos colores disponibles a elegir el mejor subconjunto de colores. Si está apagada, la tabla de colores local no está ordenada.

- Tamaño de la tabla de colores local: Si la bandera de tabla de colores local está prendida, $2^{(\text{valor de éste campo} + 1)}$ nos dará el tamaño (en bytes) de la tabla de colores local. Si no se especificó una tabla de colores local éste campo debe ser cero.

MAPA DE COLORES LOCAL

Este bloque contiene una tabla (paleta) de colores, la cual es una secuencia de tripletas de bytes representando tripletas RGB de colores:

	7 6 5 4 3 2 1 0	CAMPO	TIPO
0		Red 0	BYTE
1		Green 0	BYTE
2		Blue 0	BYTE
3		Red 1	BYTE
	... ...	...	...
764		Blue 254	BYTE
765		Red 255	BYTE
766		Green 255	BYTE
767		Blue 255	BYTE

Es usada por la imagen siguiente. Su presencia es indicada por la bandera (prendida) de tabla de colores local en el bloque descriptor de imagen, sigue inmediatamente a dicho bloque (si está presente) y contiene un número de bytes igual a 3 * 2(tamaño de la tabla de colores local + 1). Si está presente, temporalmente pasa a ser la tabla de colores activa, y será utilizada en el procesamiento de la imagen siguiente.

DATOS DE LA IMAGEN (COMPRIMIDOS Y EMPAQUETADOS EN SUB-BLOQUES)

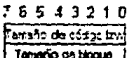
Los datos de la imagen consisten de una secuencia de sub-bloques de hasta 255 bytes cada uno:

	7 6 5 4 3 2 1 0	CAMPO	TIPO
0		Tamaño mínimo de un código LZW	BYTE
		Datos de la imagen	Sub-bloques

donde:

- Tamaño mínimo de un código LZW: Determina el número inicial de bits utilizados por códigos LZW para codificar los datos de la imagen. Cuando el número de frases detectadas por el compresor en los datos de entrada excede el número de frases codificables con el número de bits actual, el número de bits para un código LZW es incrementado en uno.

Los datos comprimidos y empaquetados de la imagen listos para almacenamiento o transmisión se pueden ver así:



Se repite tantas veces como sea necesario.

El código que termina los datos LZW-comprimidos debe aparecer antes del terminador de bloque.
Terminador de bloque

La conversión de una imagen de una serie de pixels a un flujo de caracteres a ser transmitidos o almacenados involucra varios pasos:

- Establecer el tamaño de código: Definir el número de bits necesarios para representar los datos actuales.
- Comprimir los datos: Comprimir la serie de pixels a una serie de códigos de compresión.
- Construir series de bytes: Tomar el conjunto de códigos de compresión y convertirlo a una cadena de bytes de 8 bits.
- Empaquetar los bytes: Empaquetar conjuntos de bytes en bloques precedidos por cuentas.

ESTABLECIENDO EL TAMAÑO DE CODIGO

El primer byte de los datos comprimidos es un valor que indica el número mínimo de bits requeridos para representar los pixels actuales. Normalmente será el número de bits por color. Sin embargo, debido a restricciones en el algoritmo, para imágenes en blanco y negro las cuales tienen 1 bit por color, se les asignará un tamaño de código de 2. El tamaño de código implica que los códigos de compresión deben empezar siendo un bit más largos.

COMPRESION

GIF utiliza el esquema de codificación LZW con las siguientes variaciones:

1. Es definido un código "limpiar" especial, el cual reinicializa todos los parámetros de compresión/descompresión y tablas a un estado inicial. El valor de ese código es $2^{\text{tamaño de código}}$. Por ejemplo, si el tamaño de código indicado fue 4 (imagen de 4 bits/pixel) el valor del código limpiar será $2^4 = 16 = 10000_2$. Dicho código puede aparecer en cualquier punto en la imagen y por tanto requiere que el algoritmo LZW procese los subsiguientes códigos como si se tratara del inicio de un nuevo conjunto de datos. Los codificadores deberán siempre emitir un código limpiar como el primer código de cada imagen comprimida.
2. Es definido un código "fin de información" que explícitamente indica el fin de los datos comprimidos. El procesamiento LZW termina cuando dicho código es encontrado, por lo que debe ser el último código emitido por el codificador. El valor de ese código es $2^{\text{tamaño de código}} + 1$.
3. Como ya se han definido los 2 primeros códigos ("limpiar" y "fin de información"), el primer valor disponible para un código de compresión es $2^{\text{tamaño de código}} + 2$.

4. Debido a que el esquema LZW genera códigos de longitud variable, siempre que la longitud de un código exceda la longitud de código actual (tamaño de código actual), la longitud de código será incrementada en uno. Sin embargo, dicha longitud de código será a lo más de 12 bits (lo que define un valor de código máximo de $4095 = FFF_{16}$).

   Entonces, el empaquetamiento/desempaquetamiento de los códigos debe ser alterado para reflejar la nueva longitud de código.

IMPLEMENTANDO BYTES DE 8 BITS

Debido a que el esquema utilizado crea códigos de longitud variable de entre 3 y 12 bits, esos códigos deben ser transformados a una serie de bytes de 8 bits (es decir, deben ser empaquetados en bytes), los cuales serán los caracteres a almacenar o transmitir. Para llevar a cabo lo anterior, los códigos son formados como una hilera de bits para a continuación tomar (de derecha a izquierda) conjuntos de 8 bits obteniendo así los caracteres (bytes) a ser almacenados o transmitidos. Aunque el arreglo físico del empaquetamiento cambiará con el tamaño de código, el concepto es el mismo.

EMPAQUETAMIENTO DE LOS BYTES

Una vez creados los bytes, éstos son agrupados en bloques de 0 a 255 bytes. Precediendo a cada bloque estará un byte (cuenta) indicando el número de bytes en el bloque. Un bloque con una cuenta igual a cero indica la terminación de los datos comprimidos. Por último, dichos bloques serán los datos a almacenar o transmitir para una imagen particular.

TERMINADOR GIF

Este bloque es un bloque de un solo campo que indica el fin de los datos GIF y contiene la constante 0x3B (caractor ";").

BLOQUES DE EXTENSION GIF

Hasta aquí, excepto por algunos detalles menores, el formato explicado corresponde a la versión 87a, en la cual sin embargo se había considerado ya la posible inclusión de bloques de extensión en una versión futura. Y es precisamente en la versión 89a que se adicionaron algunos bloques que extienden la definición del formato GIF.

En general, los bloques pueden ser clasificados en tres grupos: De control, de interpretación gráfica y de propósito especial. Los bloques de control (tales como la cabecera, el descriptor de pantalla lógica, el de control gráfico y el terminador), contienen información utilizada para controlar el procesamiento de los datos o información usados para establecer los parámetros del hardware. Los bloques de interpretación gráfica (tales como el descriptor de imagen y el bloque de texto), contienen información y datos utilizados para interpretar una gráfica sobre el dispositivo de despliegue. Los bloques de propósito especial (tales como los bloques de comentarios y aplicación) no son utilizados ni para controlar el procesamiento de los datos, ni para contener información o datos utilizados para interpretar una gráfica sobre el

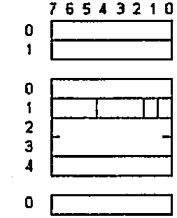
dispositivo de despliegue. Con la excepción del descriptor de pantalla lógica y la tabla global de colores, cuyo alcance es sobre todas las imágenes en el archivo, todos los demás bloques tienen alcance limitado, restringido al bloque de interpretación gráfica que les sigue.

En las secciones siguientes se explicará la sintaxis de los bloques de extensión incluidos en la versión 89a.

BLOQUE DE CONTROL GRAFICO

Este bloque (opcional) contiene parámetros utilizados al procesar un bloque de interpretación gráfica. Su alcance es solo el primer bloque de interpretación gráfica que sigue. A lo más una extensión de control gráfico puede preceder a un bloque de interpretación gráfica. Sin embargo, es posible que otras extensiones estén presentes entre éste bloque y su objetivo. Por último, éste bloque puede modificar al bloque descriptor de imagen y a la extensión de texto.

La sintaxis de éste bloque es:



CAMPO	TIPO
Introductor de extensión	BYTE
Etiqueta de control gráfico	BYTE
Tamaño del bloque	BYTE
<Campos empaquetados>	Ver abajo
Tiempo de demora	UNSIGNED
Indice de color transparente	BYTE
Terminador de bloque	BYTE

<Campos empaquetados> =		
	Reservado	3 BITS (bits 7, 6 y 5)
	Método de disposición	3 BITS (4, 3 y 2)
	Bandera de entrada de usuario	1 BIT (1)
	Bandera de color transparente	1 BIT (0)

donde:

- Introductor de extensión: Identifica el inicio de un bloque de extensión con la constante 0x21 (caracter "!").
- Etiqueta de control gráfico: Identifica el bloque como un bloque de extensión de control gráfico con la constante 0xF9.
- Tamaño del bloque: Número de bytes en el bloque, después de éste campo y hasta (pero sin incluir) el terminador de bloque. Este campo siempre contiene la constante 4.
- Método de disposición: Indica la forma en que serán tratadas las imágenes después de ser desplegadas:

0: El decodificador no tiene que tomar ninguna acción particular.
1: La imagen será dejada en su lugar.
2: Restaurar el área utilizada por la imagen al color de fondo.
3: El decodificador restaurará el área ocupada por la imagen al contenido anterior de dicha área.
4-7: A ser definidos.

- Bandera de entrada de usuario: Indica si se esperará o no una entrada por parte del usuario antes de continuar. Si está prendida, el procesamiento continuará después de la entrada del usuario (la cual puede ser un retorno de carro, pulsación de un boton del mouse, etc.). Si está apagada, no se esperará entrada por parte del usuario.
  Cuando es utilizado un tiempo de demora y la bandera de entrada de usuario está prendida, el procesamiento continuará cuando sea recibida la entrada por parte del usuario o cuando el tiempo de demora expire; lo que ocurra primero.

- Bandera de color transparente: Si está prendida, es dado un índice de color en el campo "índice de color transparente". Si está apagada, no es dado tal índice en dicho campo.

- Tiempo de demora: Si no es cero, éste campo especifica las centésimas de segundo a esperar antes de continuar con el procesamiento de los datos. El reloj empieza a correr inmediatamente después de que la imagen ha sido desplegada. Este campo puede ser utilizado conjuntamente con el campo "bandera de entrada de usuario".

- Indice de color transparente: Este índice de color es tal que cuando es encontrado, el correspondiente pixel del dispositivo de despliegue no es modificado (como si dicho pixel fuera graficado con el color transparente). Este índice está presente si y sólo si la bandera de color transparente está prendida.

- Terminador de bloque: Este bloque de longitud cero marca el fin del bloque de extensión de control gráfico. Siempre contiene la constante 0.

BLOQUE DE COMENTARIOS

La extensión (opcional) de comentarios contiene información textual la cual no es parte de la(s) imagen(es) en un archivo GIF. Es apropiada para incluir comentarios acerca de la(s) imagen(es), créditos, descripciones, o cualquier otro tipo de información no gráfica ni de control. Este bloque puede ser ignorado por el decodificador o salvado para posterior procesamiento. Bajo ninguna circunstancia éste bloque interferirá con el procesamiento de los datos.

La sintaxis de éste bloque es:

	7 6 5 4 3 2 1 0	CAMPO	TIPO
0		Introductor de extensión	BYTE
1		Etiqueta de Comentarios	BYTE
N		Comentarios	Sub-bloque
0		Terminador de bloque	BYTE

donde:

- Introductor de extensión: Identifica el inicio de un bloque de extensión con la constante 0x21 (caracter "!").

- Etiqueta de comentarios: Identifica el bloque como una extensión de comentarios con la constante 0xFE.

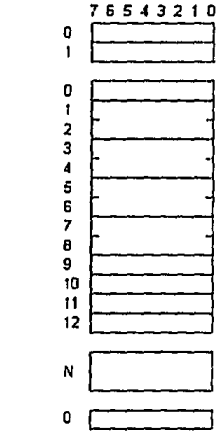
- Comentarios: Secuencia de sub-bloques, cada uno de al menos un byte y hasta 255 bytes de tamaño, indicando el tamaño en un byte precedente a los datos.

- Terminador de bloque: Este bloque de longitud cero marca el fin del bloque de extensión de comentarios. Siempre contiene la constante 0.

BLOQUE DE TEXTO

La extensión de texto contiene datos textuales y los parámetros necesarios para interpretar los datos como una gráfica. Los datos textuales serán codificados con caracteres imprimibles ASCII. Los datos serán interpretados utilizando una malla de celdas de caracteres, definida por los parámetros en los campos correspondientes. Cada caracter es interpretado en una celda individual y ocupando un solo espacio. Los caracteres son tomados secuencialmente de la porción de datos del bloque e interpretados en una celda, empezando con la celda superior izquierda en la reja y siguiendo de izquierda a derecha y de arriba a abajo. Los datos son interpretados hasta que se acaben o hasta que la reja se llene. La reja contiene un número entero de celdas; y en el caso de que las dimensiónes de la reja no sean enteras, las celdas fraccionarias se deben descartar, aunque un codificador debe ser cuidadoso de especificar precisamente las dimensiones de la reja para que ésto no ocurra. Este bloque requiere de la disponibilidad de una tabla global de colores. Este bloque es de interpretación gráfica, por tanto puede ser modificado por un bloque de control gráfico. Los bloques de texto son opcionales y cualquier número de ellos puede aparecer en el archivo.

La sintaxis de éste bloque es:



	7 6 5 4 3 2 1 0	CAMPO	TIPO
0		Introductor de extensión	BYTE
1		Etiqueta de texto	BYTE
0		Tamaño del bloque	BYTE
1		Posición izquierda de la reja	UNSIGNED
2			
3		Posición tope de la reja	UNSIGNED
4			
5		Ancho de la reja	UNSIGNED
6			
7		Altura de la reja	UNSIGNED
8			
9		Ancho de la celda	BYTE
10		Altura de la celda	BYTE
11		Indice de color del texto	BYTE
12		Indice de color de fondo	BYTE
N		Datos	Sub-Bloques
0		Terminador de bloque	BYTE

donde:

- Introductor de extensión: Identifica el inicio de un bloque de extensión con la constante 0x21 (caracter "!").
- Etiqueta de texto: Identifica el bloque como una extensión de texto con la constante 0x01.
- Tamaño del bloque: Número de bytes en la extensión después de éste campo y hasta (pero sin incluir) el inicio de la porción de datos. Este campo contiene la constante 12.
- Posición izquierda de la reja: Número en pixels de la columna del borde izquierdo de la reja, con respecto al borde izquierdo de la pantalla lógica.
- Posición tope de la reja: Número en pixels del renglón del borde superior de la reja, con respecto al borde superior de la pantalla lógica.
- Ancho de la reja: Ancho en pixels de la reja.
- Altura de la reja: Altura en pixels de la reja.
- Ancho de la celda: Ancho en pixels de cada celda en la reja.

- Altura de la celda: Altura en pixels de cada celda en la reja.

- Indice de color del texto: Indice en la tabla global de colores a ser utilizado para interpretar el texto.

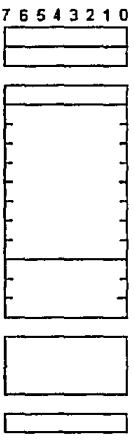
- Indice de color de fondo: Indice en la tabla global de colores a ser utilizado para interpretar el fondo del texto.

- Datos: Secuencia de sub-bloques, cada uno de al menos un byte y hasta 255 bytes de tamaño, indicando el tamaño en un byte precedente a los datos.

- Terminador de bloque: Este bloque de longitud cero marca el fin del bloque de extensión de texto. Siempre contiene la constante 0.

BLOQUE DE APLICACION

Este bloque contiene información especifica de la aplicación y su sintaxis es:



CAMPO	TIPO
Introductor de extensión	BYTE
Etiqueta de Extensión	BYTE
Tamaño del bloque	BYTE
Identificador de la aplicación	8 BYTES
Código de autenticidad de la aplicación	3 BYTES
Datos de la aplicación	Sub-Bloques
Terminador del bloque	BYTE

donde:

- Introductor de extensión: Identifica el inicio de un bloque de extensión con la constante 0x21 (caracter "!").

- Etiqueta de extensión de la aplicación: Identifica el bloque como una extensión de aplicación con la constante 0xFF.

- Tamaño del bloque: Número de bytes en la extensión después de éste campo y hasta (pero sin incluir) el inicio de la porción de datos de la aplicación. Este campo contiene la constante 11.

- Identificador de la aplicación: Secuencia de ocho caracteres imprimibles ASCII utilizada para identificar la propiedad de la extensión (identifica la aplicación que creó ésta extensión).

- Código de autenticidad de la aplicación: Secuencia de tres bytes utilizados para verificar la autenticidad del identificador de la aplicación (password).

- Terminador de bloque: Este bloque de longitud cero marca el fin del bloque de extensión de comentarios. Siempre contiene la constante 0.

TGA

La posibilidad de utilizar una computadora para hacer algo del trabajo que formalmente fue sólo del dominio de adiestrados artistas humanos ha impulsado el desarrollo de aplicaciónes gráficas. Dos de las aplicaciónes desarrolladas fueron para crear reproducciones de calidad de gráficas mecánicas y retocar digitalmente fotografías a color.

Hardware y software recientemente desarrollado hacen manejables ambos de ésos usos sin recurrir a tecnología exótica. Sin embargo, hasta no hace mucho tiempo, editar una imagen RGB de 24 bits, por ejemplo, requirió de una tarjeta de despliegue de 24 bits muy especializada. Por un largo tiempo, el hardware Truevision Targa fue la única forma práctica para trabajar con imágenes RGB, es decir, con imágenes en las cuales cada pixel es definido como un color separado, en lugar de como un número de color en una paleta. La documentación Targa llama a eso "True color", una expresión la cual ocasionalmente es aplicada a las imágenes RGB en general. En otros contextos es llamada "High color".

Las tarjetas Targa son disponibles con capacidades variables, soportando de 16 a 32 bits de color. El hardware de color especializado Targa es soportado por un formato especializado de archivo de color (el formato Targa) [TGA91], usualmente con extensión .TGA. Los archivos Targa también aparacen en Macintoshes donde tienen el tipo TPIC.

Los archivos Targa representan la forma más sofisticada de manejar colores RGB especializados (high-end). También son el formato de archivo de 24 bits disponible más ampliamente aceptado; y el archivo estándar Targa está lo suficientemente bien definido que es improbable encontrar un archivo Targa que no pueda ser leído.

Hasta recientemente los archivos Targa sólo fueron hallados en los programas de dibujo muy especializados que trabajan con hardware Targa (tal como Lumena). Sin embargo, existen varias aplicaciónes menos exóticas que trabajan con colores RGB, y el formato Targa es un medio natural para eso. Se pueden obtener archivos Targa de scanners a color y ser hallados como los archivos de trabajo de software de ray-tracing y otras aplicaciones que tratan con imágenes generadas por computadora.

El formato Targa puede contener imágenes de cualesquiera dimensiones (tan grandes como lo permita el espacio en disco para contenerlas) y con tantos colores como uno se pueda imaginar.

Por otra parte, un archivo Targa almacena una imagen bitmap junto con alguna información descriptiva opcional. El formato de archivo original contiene una cabecera, un ID de la imagen opcional, un mapa de colores si es requerido, y los datos de la imagen. El nuevo formato suma varias nuevas secciones opcionales que describen tanto la imagen como el software que la creó.

Los archivos en el nuevo formato pueden ser identificados por una firma en el nuevo pie de nota al final del archivo, descrito más adelante. Toda la nueva información viene después de la de la versión vieja, de modo que los programas que esperan leer un archivo en el formato viejo pueden leer los campos viejos de una u otra clase de archivo.

Los archivos Targa empiezan con una cabecera de tamaño fijo seguida por un ID de tamaño variable, un mapa de colores, y las secciones de la imagen. El ID de la imagen se encuentra en el offset 18, inmediatamente después de la cabecera.

Los valores que ocupan más de un byte son almacenados en el formato Intel (primero el byte menos significativo). No hay relleno o alineación de valores o secciones más allá de alineación de bytes.

Las imágenes son siempre almacenadas por renglón, pero los renglones pueden estar en orden de arriba hacia abajo o de abajo hacia arriba, y los pixels pueden ser almacenados de izquierda a derecha o de derecha a izquierda. Algunos modelos de scanners capturan una imagen en orden de renglón de arriba hacia abajo y otros modelos en orden de abajo hacia arriba.

CABECERA Y CAMPO ID

La siguiente tabla lista los campos de la cabecera:

OFFSET	LONGITUD	DESCRIPCION
0	1	Longitud del campo ID.
1	1	Tipo de mapa de colores.
2	1	Tipo de imagen.
		Mapa de colores
3	2	Primera entrada en el mapa de colores.
5	2	Longitud del mapa de colores.
7	1	Tamaño de una entrada en el mapa de colores.
		Datos de la imagen
8	2	Origen X de la imagen.
10	2	Origen Y de la imagen.
12	2	Ancho de la imagen.
14	2	Alto de la imagen.
16	1	Bits por pixel.
17	1	Bits descriptores de la imagen.

El propósito del campo ID es identificar o describir la imagen. Puede ser de hasta 255 bytes de largo. Si no existe un campo ID, como es usual, la longitud del campo ID debe ser cero.

MAPA DE COLORES

El campo "tipo de mapa de colores" en la cabecera contendrá un 1 si la imagen contiene un mapa de colores, y 0 de otra forma. Si no existe un mapa de colores, los campos correspondientes en la cabecera deben ser todos cero. Muy pocos archivos Targa contienen un mapa de colores, ya que éste fue sólo utilizado por una tarjeta de despliegue que nunca tuvo mucha aceptación.

El campo "tamaño de una entrada en el mapa de colores" contendrá los valores 15, 16, 24, o 32. Las entradas del mapa de colores de 15 y 16 bits son de la forma A RRRRR GGGGG BBBBB (representando cada letra un bit). El bit A en entradas de 16 bits es un atributo específico de la aplicación que puede ser 0.

En mapas de colores con entradas de 24 bits cada color es un byte, siendo almacenados en el archivo en el orden B, G, R. En mapas de colores con entradas de 32 bits cada color es un byte, los cuales usualmente son almacenados en el archivo en el orden B, G, R, A, donde A es un byte de atributos que puede ser descrito con más detalle en otra parte del archivo.

TIPO DE IMAGEN

El campo "tipo de imagen" describe el tipo de imagen y el esquema de compresión utilizado. La siguiente tabla lista los valores comunes de tipo de imagen (aunque otros valores han sido utilizados para otras clases de compresión), siendo los más usuales 2 y 10:

CODIGOS DE TIPO DE IMAGEN

CODIGO	DESCRIPCION
0	No hay una imagen presente.
1	Mapa de colores e imagen sin comprimir.
2	Imagen "True color" sin comprimir.
3	Blanco y negro sin comprimir.
9	Mapa de colores e imagen RLE-comprimida.
10	Imagen "True color" y RLE-comprimida.
11	Blanco y negro RLE-comprimida.

Los campos "origen X de la imagen" y "origen Y de la imagen" contienen la posición en la pantalla de la esquina inferior izquierda de la imagen. La posición (0, 0) de la pantalla es la esquina inferior izquierda de la pantalla. "Ancho de la imagen" y "Alto de la imagen" contienen el tamaño en pixels de la imagen. El número de bits por pixel usualmente es 8, 16, 24, o 32.

El byte "descriptor de imagen" contiene varios campos bits. Los cuatro bits menos significativos dan el número de bits de atributos por pixel. El significado de esos bits es definido por el campo "tipo de atributos" discutido más adelante. Los siguientes 2 bits menos significativos describen el orden en el cual los pixels son almacenados en los renglones y los

renglones en el archivo. Un valor de 00_{16} significa de izquierda a derecha y de abajo a arriba. 10_{16} significa de derecha a izquierda y de abajo a arriba. 20_{16} de izquierda a derecha y de arriba a abajo. 30_{16} de derecha a izquierda y de arriba abajo. Los dos bits más significativos han sido utilizados para indicar entrelazado de lineas raster, pero eso no es aprobado por truevision. Un valor de 00_{16} significa no entrelazado, 40 dos formas de entrelazado, y 80 cuatro formas de entrelazado.

COMPRESION RLE

Cualquier tipo de imagen puede ser RLE-comprimida. En una imagen cada renglón es comprimido de forma separada. Cada renglón comprimido es una serie de grupos, cada uno de los cuales consiste de un byte de control y uno o más pixels. El bit más significativo del byte de control indica si el grupo es "literal" o "de repetición", y los siete bits menos significativos dan la longitud del grupo. La longitud del grupo es mayor en una unidad al valor dado en el byte de control, de modo que valores del byte de control de 0_{16} a $7F_{16}$ corresponden a longitudes de 1 a 128 pixels. Si el bit más significativo es uno, el grupo es un grupo de repetición y hay un pixel que le sigue, el cual es repetido algún número de veces. Si dicho bit es 0, el grupo es un gupo literal y hay tantos pixels como indica la longitud. Por ejemplo, un byte de control de $0F_{16}$ es un grupo literal seguido por 16 pixels. Un byte de control de 88_{16} es un grupo de repetición seguido por un pixel que se repetirá 9 veces en la imagen.

CAMPOS NUEVOS

Los archivos en el nuevo formato 2.0 contienen un pie de nota, el cual identifica a los archivos que pertenecen al nuevo formato y apunta a nuevos campos opcionales que siguen a los datos de la imagen. Dichos campos pueden ser un directorio de desarrolladores, para datos especificos de la aplicación, y un área de extensión, la cual contiene un conjunto de nuevos campos que describen la imagen en varias formas. El directorio de desarrolladores y algunos campos en el área de extensión apuntan a paquetes de datos en otras partes del archivo. El orden relativo del directorio de desarrolladores, el área de extensión, y los paquetes de datos, no tiene importancia, aunque el orden convencional es los datos de los desarrolladores, el directorio de desarrolladores, el área de extensión, y finalmente el pie de nota.

PIE DE NOTA

El pie de nota (mostrado en la tabla siguiente) debe ser lo último que aparezca en el archivo. Si el archivo finaliza con la cadena "TRUEVISION-TARGA." seguida por un nulo, el pie de nota es válido.

PIE DE NOTA DE UN ARCHIVO TARGA

OFFSET	LONGITUD	DESCRIPCION
0	4	Posición en el archivo del área de extensión.
4	4	Posición en el archivo del directorio de desarrolladores.
8	17	Cadena ID "TRUEVISION-TARGA.".
25	1	Cero binario, terminador de cadena.

DIRECTORIO DE DESARROLLADORES

Las aplicaciones individuales pueden almacenar información privada en un archivo Targa. Tal información normalmente describe la imagen que contiene el archivo o algo acerca de cómo se generó.

Un programa lector puede ignorar por completo toda la información de desarrolladores y aún procesar la imagen en un archivo Targa.

Si la entrada "posición en el archivo del directorio de desarrolladores" en el pie de nota es distinta de cero, ésta proporcionará la posición del directorio de desarrolladores, mostrado en la siguiente tabla:

DIRECTORIO DE DESARROLLADORES

OFFSET	LONGITUD	DESCRIPCION
0	2	Número de entradas.
2	2	Etiqueta 1.
4	4	Posición 1.
8	4	Tamaño 1.
12	2	Etiqueta 2.
14	4	Posición 2.
18	4	Tamaño 2.
...		

Cada entrada contiene una etiqueta de 2 bytes que identifica el tipo de campo, y la posición en el archivo y tamaño del campo. El contenido del campo es proporcionado por el desarrollador que definió el campo. Truevision guarda un registro de los tipos de etiquetas.

AREA DE EXTENSION

Si el apuntador al área de extensión en el pie de nota no es cero, éste apunta a una tabla como la mostrada a continuación:

AREA DE EXTENSION

OFFSET	LONGITUD	DESCRIPCION
0	2	Tamaño del área de extensión (debe ser 495).
2	41	Nombres de los autores, texto.
43	81	Comentarios de los autores, línea 1, texto.
124	81	Comentarios de los autores, línea 2, texto.
205	81	Comentarios de los autores, línea 3, texto.
286	81	Comentarios de los autores, línea 4, texto.
367	2	Mes de creación (1 a 12).

369	2	Dia de creación (1 a 31).
371	2	Año de creación (por ej., 1991).
373	2	Hora de creación (0 a 23).
375	2	Minuto de creación (0 a 59).
377	2	Segundo de creación (0 a 59).
379	41	Nombre del trabajo, texto.
420	2	Tiempo en horas del trabajo (0 o más).
422	2	Tiempo en minutos del trabajo (0 a 59).
424	2	Tiempo en segundos del trabajo (0 a 59).
426	41	Nombre del programa de creación, texto.
467	2	100 por el número de versión.
469	1	Letra de la versión.
470	1	Color Blue de fondo o transparente.
471	1	Color Green de fondo o transparente.
472	1	Color Red de fondo o transparente.
473	1	Color Alpha de fondo o transparente.
474	2	Numerador de la tasa de aspecto de pixel.
476	2	Denominador de la tasa de aspecto de pixel.
478	2	Numerador del coeficiente Gamma.
480	2	Denominador del coeficiente Gamma.
482	4	Offset en el archivo de la tabla de corrección de color.
486	4	Offset en el archivo de la "postal" de la imagen.
490	4	Offset en el archivo de la tabla de líneas raster.
494	1	Tipo de atributo.

Los campos de texto deben terminar con un nulo. Todos los campos son opcionales. Los campos no utilizados deben ser fijados a cero, si son numéricos; o deben ser blancos o nulos si son de texto.

El campo "tipo de atributo" dá el significado de los bits de atributo en entradas de mapas de colores o pixels true color. Sus valores son: 0, no se dá el dato Alpha; 1, indefinido y puede ser ignorado; 2, indefinido y debe ser preservado; 3, datos Alpha regulares; y 4, datos Alpha premultiplicados. El código 4 significa que los valores R, G, y B en las entradas del mapa de colores o pixels han sido multiplicados por el valor de Alpha.

TABLA DE LINEAS RASTER

El área de extensión puede apuntar a una tabla de líneas raster en alguna parte del archivo. Dicha tabla es un arreglo que contiene las posiciónes en el archivo del inicio de cada renglón de los datos de la imagen. Hay una entrada por cada renglón en la imagen.

POSTAL DE LA IMAGEN

El área de extensión puede apuntar a una "postal" de la imagen, la cual es una versión en miniatura de la imagen principal. Los primeros 2 bytes de la "postal" son el ancho y alto de la imagen miniatura. En seguida de esos dos bytes están los datos de la imagen en miniatura en el mismo formato que los de la imagen principal. La "postal" no debe ser más grande que 64 x 64 pixels.

TABLA DE CORRECCION DE COLOR

La tabla de corrección de color contiene 256 valores de pixels extendidos que remapean los valores en el mapa de colores. Cada entrada en la tabla es 8 bytes de largo, consistiendo de 4 valores de 16 bits, en el orden Alpha, Red, Green, Blue. Un pixel blanco está dado por los cuatro valores hexadecimales FFFF, y un pixel negro son cuatro valores cero.

BMP

Con la creación de Windows 3, nuevos estándares de formatos de archivo fueron desarrollados, muchos de ellos reemplazando los estándares anteriores soportados por Windows 2. Además, Windows 3 tiene muchas nuevas características y facilidades.

Bajo Windows, donde todos los despliegues son gráficos por naturaleza, una utilería de captura de pantallas gráficas puede ser una herramienta básica para transferir información gráfica entre aplicaciones o simplemente salvar imágenes para su uso posterior. Al mismo tiempo, mientras bajo DOS el capturar una pantalla puede ser un procedimiento relativamente complicado, el proceso equivalente bajo Windows es considerablemente expedito y simplificado por las características inherentes a Windows.

Principal entre tales características es el Clipboard de Windows, una facilidad que proporciona almacenamiento y transferencia de información entre aplicaciones Windows. Aunque el Clipboard maneja varios tipos de información, cada uno con un formato separado, sólo las imágenes gráficas o formato bitmap es el que a nosotros atañe.

Tal formato bitmap desarrollado por Microsoft [MS90] tiene la intención de ser independiente del dispositivo, por lo que también es conocido como DIB (Device Independent Bitmap), y puede contener imágenes de 1, 4, 8, o 24 bits por pixel. Las imágenes de 1, 4, y 8 bits por pixel tienen un mapa de colores, mientras las imágenes de 24 bits están compuestas de colores directos.

Cabe mencionar que Windows 3 puede también leer archivos BMP del manejador de presentaciones 1.x de OS/2. Las dos variantes son descritas abajo.

Debido a que los archivos .BMP normalmente son utilizados en computadoras de la serie Intel 80x86, éstos usan las convenciones de Intel: El byte menos significativo almacenado en primer lugar, y sin alineación de palabras. Cada archivo contiene una cabecera de archivo, una cabecera bitmap, un mapa de colores (a menos de que la imagen sea de 24 bits), y la imagen misma. Las imágenes en el formato de archivo de Windows 3 pueden ser comprimidas utilizando el esquema RLE.

CABECERA DEL ARCHIVO

Todos los archivos DIB contienen una cabecera de archivo común:

CABECERA DE ARCHIVO WINDOWS (BITMAPFILEHEADER)

OFFSET	TAMAÑO	NOMBRE	DESCRIPCION
0	2	bfTipo	ASCII "BM".
2	4	bfTamaño	Tamaño en palabras Long (unidades de 4 bytes) del archivo.
6	2	bfReservado1	cero.
8	2	bfReservado2	cero.
10	4	bfOffBits	Offset en bytes después de la cabecera del inicio de la imagen.

El campo bfOffBits contiene la distancia en bytes del fin de la cabecera (byte 14) al inicio de los datos de la imagen, lo que facilita saltar la cabecera bitmap (que es lo que sigue a la cabecera del archivo).

Las dos variantes de los formatos bitmap pueden ser distinguidas mirando la primer palabra de la cabecera bitmap (offset 14 del archivo). Esta palabra será 12 para el formato de archivo OS/2 y 40 para el formato de archivo Windows 3.x.

CABECERA BITMAP EN WINDOWS 3

Siguiendo a la cabecera del archivo está la cabecera bitmap y, opcionalmente, el mapa de colores. La estructura de la cabecera bitmap algunas veces es referida por *BITMAPINFO*:

CABECERA DE LA IMAGEN WINDOWS (BITMAPINFOHEADER)

OFFSET	TAMAÑO	NOMBRE	DESCRIPCION
14	4	biTamaño	Tamaño de ésta cabecera, 40 bytes.
18	4	biAncho	Ancho de la imagen en pixels.
22	4	biAlto	Altura de la imagen en pixels.
26	2	biPlanos	Número de planos de la imagen, debe ser 1.
28	2	biCuentaBits	Bits por pixel, 1, 4, 8, o 24.
30	4	biCompresión	Tipo de compresión, abajo.
34	4	biTamañoImagen	Tamaño en bytes de la imagen comprimida.
38	4	biXPixelsPorMétrica	Resolución horizontal, en pixels/métrica.
42	4	biYPixelsPorMétrica	Resolución Vertical, en pixels/métrica.
46	4	biClrsUsados	Número de colores usados, abajo.
50	4	biClrsImportantes	Número de colores "importantes".
54	4 * N	bmiColores	Mapa de colores.

MAPA DE COLORES

Las imágenes que utilizan 1, 4, u 8 bits por pixel deben tener un mapa de colores. Los tamaños de los mapas de colores respectivos normalmente son de 2, 16, o 256 entradas, ya que pueden ser más chicos si la imagen no necesita un conjunto completo de colores. Si el campo biClrsUsados es distinto de cero, éste contiene el número de colores utilizados, el cual es también el número de entradas en el mapa de colores. Si tal campo es cero, el mapa es de tamaño completo. Para imágenes de 24 bits, no existe un mapa de colores ya que la imagen es de colores directos RGB.

Debido a que el dispositivo de despliegue puede no tener tantos colores disponibles como lo requiera una imagen, los colores más importantes estarán en las primeras entradas del mapa de colores. Si el campo biClrsImportantes es distinto de cero, nos indica cuales colores son importantes para una buena reproducción de la imagen.

Las entradas en el mapa de colores son de 4 bytes cada una:

ENTRADA EN EL MAPA DE COLORES WINDOWS (RGBQUAD)

OFFSET	NOMBRE	DESCRIPCION
0	rgbBlue	Valor Blue para la entrada en el mapa de colores.
1	rgbGreen	Valor Green para la entrada en el mapa de colores.
2	rgbRed	Valor Red para la entrada en el mapa de colores.
3	rgbReservado	Cero.

DATOS BITMAP EN WINDOWS 3

Los datos bitmap siguen inmediatamente al mapa de colores. Los datos pueden estar sin comprimir, o las imágenes de 4 y 8 bits pueden utilizar el esquema de compresión RLE.

Los bits son lógicamente (y físicamente en ausencia de compresión) almacenados un renglón a la vez. Cada renglón es rellenado a una frontera de 4 bytes.

BITMAPS CON UN BIT POR PIXEL

Cada pixel es un bit, empaquetando ocho en un byte. El bit más significativo en el byte es el pixel de más a la izquierda.

BITMAPS CON CUATRO BITS POR PIXEL

Las imágenes sin comprimir son empaquetadas dos pixels por byte, siendo el nibble más significativo el pixel de más a la izquierda, y rellenando cada renglón a una frontera de cuatro bytes.

Las imágenes comprimidas utilizan un formato codificado usando el esquema RLE. Dicho formato consiste de una secuencia de grupos. Hay tres clases de grupos: Grupos de repetición, Grupos literales, y Grupos especiales.

Un grupo de repetición es de dos bytes. El primer byte es una cuenta de pixels, y el segundo un par de pixels. El grupo representa el número de pixels en el primer byte, con los dos pixels en el segundo byte alternándose. Por ejemplo, los bytes hexadecimales 05 24 representan los pixels 2 4 2 4 2.

Un grupo literal es un byte cero, un byte con una cuenta de pixels, y los pixels literales. La cuenta de pixels debe ser al menos tres (uno o dos pixels pueden ser codificados como un grupo de repetición). Los pixels literales son rellenados a ceros a un número par de bytes. Por ejemplo, los bytes hexadecimales 00 05 12 34 50 00 (notar el *relleno* a un byte par) representan los pixels 1 2 3 4 5.

La secuencia especial 00 00 representa el fin de un renglón. La secuencia especial 00 01 representa el fin del bitmap. La secuencia especial 00 02 *xx yy* es una posición delta, indicando que la imagen continúa *xx* pixels a la derecha y *yy* pixels abajo.

BITMAPS CON OCHO BITS POR PIXEL

Las imágenes sin comprimir son empaquetadas un pixel por byte, rellenando cada renglón a una frontera de cuatro bytes.

Las imágenes comprimidas utilizan un formato codificado usando el esquema RLE. Dicho formato consiste de una secuencia de grupos. Hay tres clases de grupos: Grupos de repetición, Grupos literales, y Grupos especiales.

Un grupo de repetición es de dos bytes. El primer byte es una cuenta de pixels, y el segundo un pixel. Por ejemplo, los bytes hexadecimales 05 24 representan los pixels 24 24 24 24 24.

Un grupo literal es un byte cero, un byte con una cuenta de pixels, y los pixels literales. La cuenta de pixels debe ser al menos tres. Los pixels literales son rellenados a ceros a un número par de bytes. Por ejemplo, los bytes hexadecimales 00 05 12 34 56 78 9A 00 (notar el relleno a un byte par) representan los pixels 12 34 56 78 9A.

Las secuencias especiales son las mismas que para imágenes de cuatro bits.

BITMAPS CON 24 BITS POR PIXEL

Cada pixel son tres bytes, conteniendo los valores Red, Green, y Blue, en ese orden. Cada renglón es rellenado a cero a una frontera de cuatro bytes.

CABECERA BITMAP 1.x EN OS/2

El formato bitmap de OS/2 es similar a (y más simple que) el formato de Windows 3.0. La estructura de la cabecera bitmap es algunas veces referida como *BITMAPCOREINFO*:

CABECERA DE LA IMAGEN OS/2 (BITMAPCOREINFO)

OFFSET	TAMAÑO	NOMBRE	DESCRIPCION
14	4	bcTamaño	Tamaño de la cabecera, 12 bytes.
18	2	bcAncho	Ancho de la imagen en pixels.
20	2	bcAlto	Altura de la imagen en pixels.
22	2	bcPlanos	Número de planos, debe ser 1.
24	2	bcCuentaBits	Bits por pixel, 1, 4, 8, o 24.
26	3 * N	bmciColores	Mapa de colores.

MAPA DE COLORES

Las imágenes que utilizan 1, 4, u 8 bits por pixel deben tener un mapa de colores. Los tamaños de los mapas de colores son de 2, 16, o 256 entradas, respectivamente. Las entradas en el mapa de colores son de tres bytes cada una:

ENTRADA EN EL MAPA DE COLORES OS/2 (RGBTRIPLE)

OFFSET	NOMBRE	DESCRIPCION
0	rgbtBlue	Valor Blue para la entrada en el mapa de colores.
1	rgbtGreen	Valor Green para la entrada en el mapa de colores.
2	rgbtRed	Valor Red para la entrada en el mapa de colores.

DATOS BITMAP 1.x EN OS/2

Los datos bitmap siguen inmediatamente al mapa de colores. Cada renglón es rellenado con ceros a un múltiplo de cuatro bytes. La codificación de los bits es idéntica a los datos bitmap sin comprimir de Windows 3.0.

PARTE

TERCERA

DISEÑO DE PROGRAMAS DE CAPTURA Y DESPLIEGUE DE IMAGENES RGB

CAPITULO 5

CAPTURANDO UN ARCHIVO RGB EN LOS FORMATOS PCX Y GIF

Los programas de autoedición, los rastreadores ópticos, programas de dibujo, procesadores de texto, todos tienen sus propias ideas de cómo deben ser los archivos gráficos. Aunque la industria de PC's se ha "estabilizado" en un par de estándares (PCX y TIFF son los más comunes para imágenes bitmap), las imágenes que uno puede desear no siempre vendrán en esos formatos. Aún más, mucho del trabajo gráfico de hoy en día se hace en Macintosh's, y extraer un archivo con formato de Macintosh y pasarlo a una PC puede por sí solo constituir un trabajo (aunque existen programas comodines de conversión de archivos gráficos como DoDot, The Graphics Link Plus, Hijaak y Graphic Workshop).

La mayoría de las aplicaciones gráficas pueden leer y escribir en varios formatos distintos, y algunas, como PageMaker son bastante buenas tratando con múltiples formatos, pero las conversiones incluidas están limitadas en flexibilidad, especialmente cuando se trata de convertir imágenes de color a blanco y negro. Además, algunas imperfecciones de los estándares, especialmente con los archivos TIFF, pueden hacer difícil la importación de archivos aún con formatos compatibles.

El objeto de este capítulo es diseñar programas que capturarán imágenes RGB dadas en un archivo, a los formatos PCX y GIF explicados también en este trabajo y una vez convertidas a tales formatos podrán ser desplegadas utilizando las implementaciones del capítulo 6.

PCX

En el capítulo 4 se explicó ya la información contenida en la cabecera de un archivo .PCX, incluyendo la paleta de colores (para imágenes de 16 colores). Se mencionó también que dicha paleta siempre es inicializada a valores default los cuales típicamente son 0, 1, 2, 3, 4, 5, 20, 7, 56, 57, 58, 59, 60, 61, 62 y 63, y se dejo pendiente la explicación de la forma en que es obtenida la paleta para imágenes de 256 colores y el mecanismo de codificación de los datos de la imagen.

En las secciones próximas se explicarán los últimos pasos que hay que llevar a cabo para codificar una imagen RGB a formato PCX.

MAPEO DE 24 A 15 BITS POR PIXEL E HISTOGRAMA DE COLORES

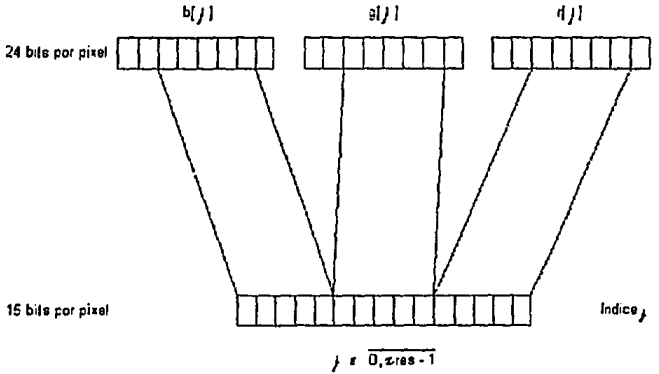
Hoy en día las imágenes digitalizadas pueden utilizar hasta 24 bits por pixel, es decir, alrededor de 16 millones de colores. El problema es que muchos monitores a color pueden desplegar sólo 256 colores a la vez, por lo que para adaptarse a tales despliegues es necesario un mecanismo de elección de los 256 colores que mejor representen todas las sutiles intensidades de colores de la imagen y a continuación mapear esas miles de intensidades en los 256 colores elegidos (notar que esto constituye una precuantificación de los datos, aunque el propósito no es precisamente comprimirlos). Un ejemplo de un despliegue tal es la tarjeta VGA en la cual los pixels (de 8 bits) en el frame buffer no son interpretados como colores, sino como índices a una pequeña paleta de 256 colores de alguna más grande determinada por el hardware. Dicha pequeña paleta normalmente es cargada a una tabla de la VGA quien traduce un índice de 8 bits al correspondiente color de despliegue. Sin embargo, al tener nosotros una imagen de 24 bits por pixel, lo que necesitamos para desplegarla es traducir tales pixels a índices (de 8 bits) para obtener una forma apropiada para el frame buffer.

Entonces (dado que aquí trabajaremos con la VGA), necesitamos dada una imagen de 24 bits, elegir una pequeña paleta de 256 colores que mejor represente el rango completo de colores en la imagen y para desplegarla traducir los colores de 24 bits a índices de paleta de 8 bits.

Para obtener una paleta de los 256 colores más representativos en la imagen, primero inicializamos un histograma a ceros, para enseguida recorrer pixel por pixel la imagen convirtiendo cada PIXEL de 24 bits A un INDICE de 15 bits (la razón de elegir 15 bits será clara más adelante), incrementando por último la entrada correspondiente en el histograma. Más detalladamente, dados tres arreglos r[], g[] y b[] de bytes:

1 Inicializar Histograma[] a ceros
2 En los arreglos r[], g[] y b[] se leen cada uno de los tres planos de color (RRR..., GGG..., y BBB..., respectivamente) de la i-ésima línea raster (recordar que una línea raster en el archivo RGB viene dada por 3 sublíneas que son los planos de color).
3 A continuación se toman los bits del 1 al 5 (mapeo de 24 a 15 bits por pixel) de las entradas correspondientes de dichos arreglos y se combinan para formar un índice; los 5 bits menos significativos del cual serán los

correspondientes al rojo, los siguientes 5 bits menos significativos los correspondientes al verde, y los siguientes 5 bits menos significativos los correspondientes al azul:



4 Se incrementa en uno la entrada correspondiente al índice obtenido en el histograma (si una entrada ya llegó a 255 no se incrementará más):

Histograma[Indice$_j$] <-- Histograma[Indice$_j$] + 1 con $j \in \{0, ..., xres - 1\}$.

Los pasos 1 a 4 se llevan a cabo para cada $i \in \{0, ..., yres - 1\}$.

El fragmento de código C correspondiente es:

```
for(j = 0; j < 32768; j++)
   Histograma[j] = 0;
fread(&r[0], sizeof(char), xres, arch_RGB);
fread(&g[0], sizeof(char), xres, arch_RGB);
fread(&b[0], sizeof(char), xres, arch_RGB);
for(indice = 0; indice < xres; indice++)
{
   Color.Red    = (r[indice] & 0x3E);
   Color.Green  = (g[indice] & 0x3E);
   Color.Blue   = (b[indice] & 0x3E);
```

```
    color_no = (Color.Red >> 1) | (Color.Green << 4) | (Color.Blue << 9);
    if(Histograma[color_no] < 255)
      Histograma[color_no]++;
}
```

Existen varios puntos a aclarar. Si tomáramos directamente los 24 bits como índice, necesitaríamos un histograma con 2^{24} entradas que pudieran almacenar una palabra (es decir, necesitaríamos una tabla de 32 MB). Por tanto, para salvar espacio en memoria se utiliza un histograma de 2^{15} (32 KB) entradas. Las variables x_{res} y y_{res} son las dimensiones de la ventana de despliegue, es decir, corresponden a la entrada "dimensiones de la ventana" de la cabecera (bytes 4 a 11) y deben ser proporcionadas como información de entrada en el archivo RGB. Normalmente dichas dimensiones concuerdan con las resoluciones vertical y horizontal (entradas 12 a la 15 de la cabecera) del dispositivo de despliegue. Por último, el efecto de mapear un pixel de 24 bits a uno de 15 bits (precuantificación) tiene el efecto de reducir el número de colores diferentes e incrementar la frecuencia de cada color, y el establecer una frecuencia tope de 255 implica una distribución uniforme de los colores en la imagen (la cual dicho sea de paso no es la mejor). Para comprender mejor la teoría del proceso descrito se puede consultar el artículo de Paul Heckbert "COLOR IMAGE QUANTIZATION FOR FRAME BUFFER DISPLAY" [Hec82], o el de Dave Pomerantz "A FEW GOOD COLORS" [Pom90].

ORDENAMIENTO DE LOS COLORES

La experiencia ha mostrado que de los 32 KB colores dados por los índices del histograma muchas imágenes utilizan alrededor de 5000, y es en éstos últimos en los que debemos centrar nuestra atención. Para ésto creamos una Lista de colores que contendrá sólo aquellos índices de 15 bits de los colores que ocurrieron en la imagen para finalmente elegir de entre éstos los 256 colores más representativos.

Una vez construido el histograma de colores éste es inspeccionado, y al ser encontrada una entrada con un valor distinto de cero, dicho valor es guardado en un arreglo de Frecuencias y la entrada en un arreglo Lista_de_colores:

```
ultimo_color_hallado = -1;
for(j = 0; j < 32768; j++)
{
   if(Histograma[j] > 0)
   {
      ultimo_color_hallado++;
      Lista_de_colores[ultimo_color_hallado] = j;
      Frecuencias[ultimo_color_hallado] = Histograma[j];
   }
}
sort(0, ultimo_color_hallado);
```

Entonces, los 256 colores más representativos serán los más frecuentes, por lo que para hallarlos ordenamos el arreglo de Frecuencias (última línea del código anterior) de *mayor* a menor (mediante algún algoritmo de ordenamiento, en éste caso el Quicksort), y con ésto tendremos en sus 256 primeras entradas los 256 colores más representativos.

Si recordamos que el arreglo Lista_de_colores contiene índices, los cuales son los pixels de 24 bits originales cuantificados a 15 bits, y cuyas frecuencias están dadas en las entradas correspondientes del arreglo de Frecuencias (antes del ordenamiento), entonces, los cambios hechos al ordenar el arreglo de frecuencias deben también ser efectuados sobre el arreglo Lista_de_colores. Al final del ordenamiento tenemos que:

$Frecuencias[i] \geq Frecuencias[j]$ o $Histograma[Lista\_de\_colores[i]] \geq Histograma[Lista\_de\_colores[j]]$ si $i \leq j$

MODIFICACION DE LOS COLORES EXTRAS

Dado que hemos ya identificado los 256 colores más frecuentes en la imagen, el siguiente paso es mapear todos los colores que ocurrieron en ésta a dichos 256. Para hacer lo anterior debemos definir un mapeo basado en algún criterio, y uno que estadisticamente es bueno es el criterio de mínimos cuadrados, por lo que es el que se utilizará.

A partir de la localidad 256 y hasta la localidad dada por el número de colores encontrados en la imagen (último valor asumido por último_color_hallado) del arreglo Lista_de_colores se encuentra para cada una de dichas localidades i el valor de la localidad $j \in \{0, \ldots, 255\}$ que satisfaga que la cantidad $m_i = (R_i - R_j)^2 + (G_i - G_j)^2 + (B_i - B_j)^2$ sea mínima,o lo que es lo mismo, se busca el color que más se asemeje (de acuerdo al criterio de los mínimos cuadrados) al color en la localidad i de entre los 256 colores elegidos.

Por otra parte, ya que las frecuencias de los colores nos sirvieron únicamente para seleccionar la paleta, numeramos los colores en el arreglo Frecuencias, es decir, para $l \in \{0, \ldots, 255\}$ $Frecuencias[l] \leftarrow l$. Esto con el objeto de poder realizar el mapeo de todos los colores ocurridos en la imagen en los 256 seleccionados:

Al ser encontrada j tal que minimiza la cantidad m_i, el mapeo es dado por $Frecuencias[i] \leftarrow j$, es decir, **al pixel *i*-ésimo se le asignará el color *j*-ésimo** reemplazando su frecuencia por el número j de color. El fragmento C correspondiente es:

```
for(i = 0; i < 256; i++)
   Frecuencias[i] = i;
for(i = 256; i <= ultimo_color_hallado; i++)
{
   d1 = 32768;
   for(j = 0; j < 256; j++)
   {
      Color.Red   = (Lista_de_colores[i] << 1) & 0x3E;
      Color.Green = (Lista_de_colores[i] >> 4) & 0x3E;
      Color.Blue  = (Lista_de_colores[i] >> 9) & 0x3E;
      Color2.Red  = (Lista_de_colores[j] << 1) & 0x3E;
      Color2.Green= (Lista_de_colores[j] >> 4) & 0x3E;
      Color2.Blue = (Lista_de_colores[j] >> 9) & 0x3E;
```

```
        mi =   (Color.Red - Color2.Red) * (Color.Red - Color2.Red) +
               (Color.Green - Color2.Green) * (Color.Green - Color2.Green) +
               (Color.Blue - Color2.Blue) * (Color.Blue - Color2.Blue);
        if(mi < d1)
        {
          d1 = mi;
          d2 = j;
        }
      }
      Frecuencias[i] = d2;
  }
```

Entonces, una vez hecho lo anterior el mapeo M:(número de color de 0 a 255) —> (pixels cuantificados) queda definido por M:Frecuencias[i] —> Lista_de_colores[i] con i ∈ {0, ... , ultimo_color_hallado}.

Por otra parte, ya que en sus primeras 256 entradas Lista_de_colores[] contiene los 256 pixels (cuantificados) más frecuentes en la imagen, éstos constituyen la paleta elegida, por lo que es guardada en un arreglo Paleta_elegida[][]:

```
for(j = 0; j < 256; j++)
{
   Color.Red     = (Lista_de_colores[j] << 1) & 0x3E;
   Color.Green   = (Lista_de_colores[j] >> 4) & 0x3E;
   Color.Blue    = (Lista_de_colores[j] >> 9) & 0x3E;
   Paleta_elegida [j][0] = Color.Red;
   Paleta_elegida [j][1] = Color.Green;
   Paleta_elegida [j][2] = Color.Blue;
}
```

CODIFICACION DE LA IMAGEN UTILIZANDO EL ESQUEMA RLE

Hasta aquí hemos numerado en el arreglo Frecuencias[] los 256 colores más representativos contenidos en el arreglo Lista_de_colores y realizado el mapeo guardando un número de color (de 0 a 255) en las restantes entradas de Frecuencias[], color con el cual será graficado el pixel en la entrada correspondiente en el arreglo Lista_de_colores.

Es importante resaltar que dicha numeración constituye la traducción de los pixels de 24 bits a índices de 8 bits propios para el Frame Buffer, los cuales por otra parte deberán apuntar a la paleta seleccionada.

De acuerdo a lo anterior, la codificación de los datos de la imagen será llevada a cabo traduciendo los pixels de 24 bits a su color de mapeo correspondiente y codificando éste último valor. Haciendo las cosas de ésta forma, el decodificador no tendrá mas que cargar la paleta seleccionada y el Frame Buffer con los datos decodificados (sin hacerles ninguna operación adicional) para poder desplegar la imagen.

En éste punto es útil reemplazar las entradas del Histograma por números de color (al igual que se hizo con el arreglo Frecuencias[])

```
/* primero inicializamos el Histograma a ceros */
for(j = 0; j < 32768; j++)      Histograma[j] = 0;
for(j = 0; j <= ultimo_color_hallado; j++)
```

```
/* traducción de un pixel cuantificado a su color de mapeo correspondiente */
   Histograma[Lista_de_colores[j]] = Frecuencias[j];
```

para traducir un pixel de 24 bits a su color de mapeo correspondiente.

Hecho lo anterior estamos en condiciónes de codificar los datos de la imagen, para lo cual nos volvemos a posicionar al inicio del archivo RGB.

Como ya mencionamos, el esquema de codificación utilizado para comprimir las imágenes PCX es el RLE, el cual codifica una corrida mayor que uno de un símbolo como un par cuenta-símbolo y un símbolo con longitud de corrida 1 es escrito tal cual. Para efectos de que el decodificador distinga entre un dato cuenta y un dato símbolo, se ha convenido en que la cuenta se identifique con la bandera 0xC, es decir, la cuenta estará formada por la bandera C y la cuenta misma, de esta forma si un símbolo x se repite n veces se codificará como Cnx. Por otro lado, dado que los datos normalmente son leídos byte a byte, la cuenta junto con su bandera deberán estar almacenados en justamente un byte, por lo que si la bandera la situamos al extremo izquierdo del byte, ésta prenderá los dos bits más significativos, es decir, $C0_{16}$ = 11000000_2. De ésta forma tenemos 6 dígitos binarios para almacenar la longitud de una corrida, es decir, se podrá codificar una corrida con longitud máxima de $2^6 - 1 = 63$.

Obviamente existen números de color que tienen sus dos bits más significativos prendidos, y para no confundirlos con un byte cuenta se les debe dar un tratamiento especial. Dicho tratamiento consiste en que, independientemente de su longitud, tales números de color siempre se codificarán como un par cuenta-símbolo; de ésta forma siempre que el decodificador encuentre un byte con sus dos bits más significativos prendidos no habrá duda de que se trata de una cuenta.

Por último, la codificación siempre se lleva a cabo una línea a la vez, es decir, siempre que empieza una nueva línea raster todas las variables son inicializadas.

A continuación se dá el fragmento de código C correspondiente a la codificación de una línea raster:

```
fread(&r[0], sizeof(char), xres, arch_RGB);
fread(&g[0], sizeof(char), xres, arch_RGB);
fread(&b[0], sizeof(char), xres, arch_RGB);
Color.Red   = (r[0] & 0x3E);
Color.Green = (g[0] & 0x3E);
Color.Blue  = (b[0] & 0x3E);
color_no = (Color.Red >> 1) | (Color.Green << 4) | (Color.Blue << 9);
cuenta = 1;
col_ultimo = Histograma[color_no];
for(indice = 1; indice <= xres; indice++)
{
   Color.Red     = (r[indice] & 0x3E);
   Color.Green   = (g[indice] & 0x3E);
   Color.Blue    = (b[indice] & 0x3E);
   color_no = (Color.Red >> 1) | (Color.Green << 4) | (Color.Blue << 9);
   if(indice == xres)
      col_actual = col_ultimo - 1;
   else
```

```
        col_actual = Histograma[color_no];
      if((col_actual == col_ultimo) && (cuenta < 63))
        cuenta++;
      else
      {
        sal_cuenta = ((unsigned char) cuenta | 0xC0);
        if(((col_ultimo & 0xC0) == 0xC0) || (cuenta > 1))
          fputc(sal_cuenta, arch_pcx);
        fputc(col_ultimo, arch_pcx);
        col_ultimo = col_actual;
        cuenta = 1;
      }
    }
```

ESCRITURA DE LA PALETA DE COLORES ELEGIDA PARA LA IMAGEN

El último paso para codificar una imagen de 256 colores a formato PCX es escribir la paleta de colores seleccionada. Para que el decodificador pueda verificar que una paleta tal existe, se escribe antes de ésta el número 0xC, por lo que el decodificador puede verificar la existencia de la paleta viendo primero en el byte 1 de la cabecera que se trate de la versión 3.0 (en dicho byte debe estar el número 5), y a continuación regresarse 769 bytes del fin de archivo y leer el byte en esa posición; si dicho byte es 0xC, sigue la paleta; de lo contrario existe un error de datos.

A continuación siguen las líneas finales del código C que codifica una imagen RGB a formato PCX:

```
fputc(0x0C, arch_pcx);
for(i = 0; i < 256; i++)
  for(j = 0; j < 3; j++)
    fputc(Paleta_elegida[i][j] * 4, arch_pcx);
fclose(arch_pcx);
fclose(arch_RGB);
```

GIF

De la explicación en el capítulo 4 del formato GIF podemos notar que tal formato es orientado a manejar índices de color en lugar de colores RGB, por lo que es más efectivo con modos de 256 colores a diferencia de, por ejemplo PCX o IMG, los cuales son más efectivos con modos de 16 colores. Por otra parte, los codificadores GIF tienden a ser un poco más lentos (aunque los decodificadores GIF van a la par con otros métodos) sacrificando así rapidez de codificación en pro de poco espacio de almacenamiento, lo cual para efectos de almacenamiento o transmisión es más importante.

En las secciónes siguientes se explicarán los pasos para convertir un archivo RGB a formato GIF (versión 87a).

CABECERA GIF Y DESCRIPTOR DE PANTALLA LOGICA

En el programa considerado aquí se leerán imágenes RGB de 24 bits con 320 * 200 pixels que se mapearán en 256 colores y se codificarán con la versión GIF87a. Además, siempre se proporcionará un mapa de colores global y no se dará información sobre la tasa de aspecto de pixel. Por lo que el fragmento de código C para escribir la información de la cabecera y el descriptor de pantalla lógica (una vez abiertos los archivos RGB y de salida) es:

```
fread(&xres, sizeof(int), 1, arch_RGB);
fread(&yres, sizeof(int), 1, arch_RGB);
strcpy(cabecera_gif.nombre,"GIF");
strcpy(cabecera_gif.version,"87a");
cabecera_gif.xres = xres;
cabecera_gif.yres = yres;
cabecera_gif.empaquetado = 0xF7;   /* M = 1; cr = 7; pixel = 7 */
bits = 8;
colores = 256;
cabecera_gif.color_de_fondo = 0;
cabecera_gif.tasa_asp_pixel = 0;
fwrite(&cabecera_gif,1,13,arch_GIF);
```

en donde cabecera_gif es una estructura definida por:

```
struct
{
  char nombre[3];
  char version[3];
  int xres,yres;
  unsigned int empaquetado;
  char color_de_fondo;
  char tasa_asp_pixel;
} cabecera_gif;
```

Con excepción de la variable bits, todo el código es autoexplicativo. Recordando que el primer byte del bloque de datos comprimidos es el tamaño mínimo de un código LZW y que éste normalmente está dado por el número de bits por componente de color (resolución de color), número el cual es precisamente el almacenado en el subcampo cr del campo

empaquetado en la anterior estructura incrementado en 1, entonces es útil en éste punto guardar (en la variable bits) dicha información.

MAPA DE COLORES GLOBAL

Para determinar el mapa global de colores se llevan a cabo EXACTAMENTE los mismos pasos (Mapeo de 24 a 15 bits por pixel e Histograma de colores, Ordenamiento de los colores y modificación de los colores extras) que para el caso del formato PCX, y por tanto el código C correspondiente es el mismo.

En éste caso, la única diferencia es que aquí si se escribe la paleta elegida inmediatamente después de ser determinada (recordar que en el formato PCX una paleta de 256 colores es siempre pasada al final del archivo porque antes no hay un lugar donde pueda ser almacenado), es decir, el proceso de seleccion de la paleta global termina escribiéndola al archivo GIF:

```
for  (j = 0 ; j < 256 ; j++)
  {
    Paleta_elegida[j].red = ((Lista_de_colores[j] << 1 ) & 0x3E) << 2;
    Paleta_elegida[j].green= (( Lista_de_colores[j] >> 4) & 0x3E) << 2;
    Paleta_elegida[j].blue = (( Lista_de_colores[j] >> 9 ) & 0x3E) << 2;
  }
  fwrite(&Paleta_elegida,1,768,arch_GIF);
```

DESCRIPTOR DE IMAGEN

La única imagen que se almacenará en el archivo GIF iniciará en la posición (0, 0) de la pantalla lógica, teniendo como ancho xres pixels, como alto yres pixels, usará el mapa global de colores y su interlacing será el del dispositivo en el que se despliegue. Por lo que el fragmento de código C correspondiente es:

```
fputc(',',arch_GIF);              /* Caracter separador de imagen                    */
fputc(0,arch_GIF);                /* El byte menos significativo de esta palabra con-      */
fputc(0,arch_GIF);                /* tiene el pixel de inicio de la izq de la pantalla*/
fputc(0,arch_GIF);                /* El byte menos significativo de esta palabra con-*/
fputc(0,arch_GIF);                /* tiene el pixel de inicio del tope de la pantalla*/
fwrite(&xres,1,2,arch_GIF);       /* Ancho de la imagen en pixels (byte MS)*/
fwrite(&yres,1,2,arch_GIF);       /* Alto de la imagen en pixels (byte MS) */
fputc(0,arch_GIF);                /* SE USA EL MAPA DE COLORES GLOBAL (M = 0) Y LA
                                  IMAGEN ES FORMATEADA EN ORDEN SECUENCIAL (I = 0) */
```

DATOS COMPRIMIDOS DE LA IMAGEN

Recordando que al término del proceso de seleccion de la paleta global en el arreglo Lista_de_colores[] quedaron los pixels (cuantificados a 15 bits) que ocurrieron en la imagen, y que en el arreglo Frecuencias[] quedaron los índices de color (números enteros entre 0 y 255) con que serán graficados dichos pixels (éste proceso es explicado en detalle para el

formato PCX en la sección 5.1), para que al tiempo de codificar también realizemos el mapeo del pixel leído a su correspondiente índice de color (aproximado) es conveniente reutilizar el arreglo Histograma[], es decir, ya que la cuantificación de un pixel de 24 bits nos lleva a un índice de 15 bits del Histograma[], es útil tener en ésa entrada el índice de color (aproximado) que le corresponde al pixel en cuestión. Lo anterior se traduce en el código:

```
for(j = 0; j < 32768; j++)
  Histograma[j] = 0;
for(j = 0; j <= ultimo_color_hallado; j++)
  Histograma[Lista_de_colores[j]] = Frecuencias[j];
```

Como ya se mencionó en la descripción del formato, los datos de la imagen son comprimidos utilizando el esquema LZW, el cual está basado en una tabla (diccionario) de cadenas de pixels presentes en la imagen y cuyos índices de entrada (apuntadores a las cadenas) son los códigos para tales cadenas. Sin embargo, para efectos prácticos, las 2 primeras entradas de dicha tabla son ocupadas por los 2 códigos especiales terminador = $2^{\text{tamaño de código}}$ y fin_info = terminador + 1, en donde: terminador reinicializa todos los parámetros de compresión/descompresión y fin_info marca el fin de los datos comprimidos.

Por otra parte, el bloque de datos comprimidos empieza con un byte conteniendo el número inicial de bits menos 1 utilizados por códigos LZW (bitsinicio - 1):

```
rewind(arch_RGB);
bitsinicio = bits + 1;
terminador = 1 << (bitsinicio - 1);      /* 2 ^ (Tamaño de Codigo)          */
fin_info = terminador + 1;               /* fin_info == codigo terminador + 1  */
fputc(bits,arch_GIF);                    /* El primer caracter de un flujo de datos GIF es
                                         un valor indicando el numero mínimo de bits requeridos para representar el conjunto de valores de -
                                         un pixel actual,normalmente=No de bits por color*/
```

La sentencia rewind aquí es porque cuando terminamos el proceso de seleccion de la paleta global nos quedamos apuntando al fin del archivo RGB.

Ya que los códigos son cadenas de bits de longitud variable, el formato especifica el agrupamiento de tales cadenas de bits en bytes, los cuales a su vez son empaquetados en bloques de hasta 255 bytes. El siguiente fragmento son inicializaciónes para efectos del empaquetamiento el que será explicado más adelante.

```
codigobits = bitsinicio;
nbytes = 0;
nbits = 0;
for (i = 0; i < 256; i++)
  bloque[i] = 0;
inicializa();              /* INICIALIZAR LOS PARAMETROS DE COMPRESION */
```

Realmente, la última línea es la que entre otros parámetros inicializa la tabla de cadenas.

Es en éste punto cuando la codificación propiamente dicha da comienzo:

El algoritmo utilizado es el algoritmo LZW básico (explicado en la parte I) con la utilización adicional de una técnica Hash para manejar la tabla de cadenas dinámicamente en memoria.

En ésta técnica, la clave "codigoHash" para una cadena prefijoK[] de longitud mayor que 1 queda determinada por la longitud de ésta y por los índices de color que la constituyen:

$$\text{codigoHash} = \begin{cases} [301 * (\text{prefijoK}[1] + 1)] * (K + \text{longitud}) & \text{si longitud} = 2 \\ \text{codigoHash} * (K + \text{longitud}) & \text{si longitud} > 2 \end{cases}$$

en donde K es el último índice de color en la cadena, prefijoK[0] contiene la longitud de la cadena y prefijoK[i] contiene el i-ésimo índice de color en la cadena.

La función Hash por otra parte es:

entradaHash = (codigoHash + 1) mod 5003

Por último, al ocurrir una colisión en la búsqueda de una entrada Hash, se busca una entrada secundaria por inspección lineal:

entradaHash <-- (entradaHash + 1) mod 5003.

El siguiente fragmento de código C corresponde al final de la función salvaGIF, la cual salva la imagen RGB a formato GIF:

```
fread(&xres,sizeof(int) ,1,arch_RGB);
fread(&yres,sizeof(int),1,arch_RGB);
while(!feof(arch_RGB))
{
  fread(&linearaster,sizeof(int),1,arch_RGB);
  if (linearaster >= yres)
  {
    printf("Faltan datos en el archivo. \n");
    exit(1);
  }
  fread(&r[0], sizeof(char), xres, arch_RGB);
  fread(&g[0], sizeof(char), xres, arch_RGB);
  fread(&b[0], sizeof(char), xres, arch_RGB);
  for (col = 0; col < xres; col++)
  {
    K = Histograma[((r[col] & 0x3E) >> 1) | ((g[col] & 0x3E) << 4) | ((b[col] & 0x3E) << 9)];      /* P A S O */
```

```
prefijoK[0] = ++longitud;
prefijoK[longitud] = K;
switch(longitud)
{
 case 1:
       prefijo = K;                                  /* Cadena prefijo <- primer caracter... */
        break;                                       /* leido de la entrada (solo al inicio del proceso*/
 case 2:
       codigoHash = 301 * (prefijoK[1] + 1);
 default:
        codigoHash *= (K + longitud);                /*     clave K     */
        entradaHash = ++codigoHash % 5003;           /* H(K) = indice   */

        /* BUSQUEDA DE LA ENTRADA HASH Y MANEJO DE COLISIONES */

        for (i = 0; i < 5003; i++)
        {

         /* Se obtiene la próxima entrada por INSPECCION LINEAL */
         entradaHash = (entradaHash + 1) % 5003;

         if (memcmp(&TABLA[indice_cadena[entradaHash] + 2], prefijoK,longitud + 1) == 0)
           break; /* Se encontro que prefijoK esta en la TABLA y por tanto se prepara para repetir PASO */
         if (indice_cadena[entradaHash] == 0)
           i = 5003;                                 /* Se encontro una entrada libre en TABLA */
        }
        if (indice_cadena[entradaHash] != 0 && longitud < 97)
        {
         memcpy(&prefijo,&TABLA[indice_cadena[entradaHash]],2);
         break;                  /*    prefijo <- prefijoK y repetir PASO    */
        }
       /*       DE LO CONTRARIO       */
       escribe_codigo(prefijo);          /* arch_GIF <- codigo(prefijo) */
       entradas++;
       if (indice_cadena[entradaHash] == 0)
       /* TABLA <- TABLA + prefijoK                  */
       {
        temp = entradas + fin_info;
          indice_cadena[entradaHash] = proximo;
       /*   Escribir en la TABLA el No de entrada, ...     */
        memcpy(&TABLA[proximo],&temp,2);
       /*...la longitud de la cadena; asi como la cadena misma*/
        memcpy(&TABLA[proximo + 2],prefijoK,longitud + 1);
        proximo += longitud + 3;
        actual++;
       }
       /* prefijo <- K (proximas cuatro instrucciones)      */
       prefijoK[0] = 1;
       prefijoK[1] = K;
       longitud = 1;
       prefijo = K;
       if ((entradas + fin_info) == (1 << codigobits))
      /* Siempre que la longitud de un código LZW exceda   */
      /* la longitud de código actual, ésta se debe incrementar en 1*/
         codigobits++;
       if (entradas + fin_info > 4093 || actual > 3335 || proximo > 15379)
      /* Si la longitud de código actual es 12 ya no se incrementará más, y se reinicia la codificación*/
```

```
            /* en el punto donde se requiere éste incremento como si se tratára de un nuevo conjunto de datos */
            {
              escribe_codigo(prefijo);
              inicializa();
            }
          }                    /* fin Default */
        }                      /* fin Switch */
      }                        /* fin for REPETIR PASO*/
    }                          /* fin While */
/*  Si no existe tal K arch_GIF <- codigo(prefijo) y terminar      */
escribe_codigo(prefijo);
escribe_codigo(fin_info);
fputc(0,arch_GIF); /*TERMINA CODIFICACION LZW*/
fputc(';',arch_GIF); /* TERMINADOR GIF */
fclose(arch_GIF);
fclose(arch_RGB);
}
```

Dos funciones adicionales son utilizadas en éste código, inicializa() y escribe_codigo().

Como ya se mencionó anteriormente, la función inicializa() tiene como propósito reinicializar los parámetros de compresión:

En primer lugar, dicha función debe (de acuerdo a la especificación del formato) escribir el código terminador al archivo GIF, para a continuación inicializar la tabla de cadenas y por último restablecer el tamaño mínimo de un código LZW a su valor inicial (bitsinicio):

```
void inicializa(void)
{
  escribe_codigo(terminador);
  entradas = 0;
  actual = 0;
  proximo = 1;
  longitud = 0;
  codigobits = bitsinicio;
  TABLA[0] = NULL;
  memset(indice_cadena,0x00,10000);
}
```

en donde la variable entradas nos sirve para determinar el punto en el que el tamaño de código se debe incrementar en 1, actual y próximo nos sirven para determinar el punto donde se deben reinicializar los parámetros de compresión, e indice_cadena contendrá los apuntadores a las frases en el diccionario.

Por último, la función escribe_codigo() agrupa los códigos que recibe a la derecha y los empaqueta en bloques de bytes, los cuales son los datos escritos al archivo GIF:

```
void escribe_codigo(unsigned int codigo)
{
  bloque[nbytes] |= ((codigo << nbits) & 0xFF);
```

```
  bloque[nbytes + 1] |= ((codigo >> (8 - nbits)) & 0xFF);
  bloque[nbytes + 2] |= (((codigo >> (8 - nbits)) >> 8) & 0xFF);
  nbits += codigobits;
  while (nbits >= 8)
  {
    nbits -= 8;
    nbytes++;
  }
  if (nbytes < 251 && codigo != fin_info)
    return;
  if (codigo == fin_info)
    while (nbits > 0)
    {
        nbits -= 8;
        nbytes++;
    }
  fputc(nbytes,arch_GIF);
  fwrite(bloque,nbytes,1,arch_GIF);
  memcpy(bloque,&bloque[nbytes],5);
  memset(&bloque[5],0x00,250);
  nbytes = 0;
}
```

CAPITULO 6

DESPLEGANDO IMAGENES EN LOS FORMATOS IMG, PCX Y GIF

Sería ideal que todos los formatos de imágenes gráficas disponibles pudieran ser leídos por todas las aplicaciones que podrían utilizarlos. Sin embargo ese no es el caso, ya que aún en un solo formato existen a veces tantas variaciones que una aplicación que lo soporte puede encontrarse con un archivo de tal formato que no pueda desplegar. Es más factible que mediante un programa se reciba una imagen en algún formato como entrada y se dé como resultado la misma imagen en algún otro formato (aunque estos programas también tienen sus problemas).

En las secciones siguientes se describirá el algoritmo para desplegar las imágenes capturadas en cada uno de los formatos (y con los programas) descritos en el capítulo 5.

IMG

De acuerdo a la información dada en la cabecera de un archivo .IMG, es posible que la imagen ahí almacenada tenga una resolución de hasta 65536 X 65536 pixels (la cual es mayor que cualesquier resolución en los modos de despliegue que son manejados por la VGA). Por lo que para desplegar un archivo .IMG primero veremos si su resolución es la misma que la resolución de alguno de los modos estándares de la EGA/VGA, y de ser así, utilizaremos dicho modo de despliegue. De lo contrario, seleccionaremos la resolución de despliegue más alta disponible para la VGA en 16 colores (640 X 480 pixels), indicaremos la resolución del archivo, y el usuario deberá proporcionar el renglón y columna a partir de los cuales se desplegará la parte correspondiente de la imagen.

El corazón del programa de despliegue es la función restaura_pantalla() la cual, una vez abierto el archivo .IMG, checa los primeros 5 bytes de la cabecera y si no concuerdan con los de un archivo .IMG despliega el mensaje apropiado y termina. En seguida se checan las resoluciónes max_col y max_renglon y si max_col > 8 KB (Se manejarán archivos con una resolución máxima de 8 KB), se manda el mensaje apropiado y termina el programa. Si la resolución max_renglon >= 480, se selecciona el modo 18 (640 X 480 pixels), si no, se selecciona el modo 16 (640 X 350 pixels). A continuación, si alguna de las resoluciones (max_col o max_renglon) es distinta que su correspondiente del modo seleccionado (xres y yres), se escriben en pantalla el ancho y alto de la imagen y del dispositivo y se pide al usuario el punto a partir del cual será desplegada la imagen. Enseguida se leen los próximos 16 KB del archivo (o hasta alcanzar el fin de archivo), se fija el modo de despliegue, se fijan los registros de la paleta EGA para mostrar los colores apropiados de acuerdo al metodo 1, y se determina el punto hasta donde será desplegada la imagen.

La función entra entonces a un loop While que itera una vez para cada renglón en el archivo (notar que independientemente del punto de inicio de despliegue, se procesan todos los datos del archivo, cambiando sólo lo que se despliega). A continuación, se buscan los códigos para los distintos casos (de duplicación de renglónes, repetición de patrones, cadenas de caracteres y de corridas sólidas) dirigiendo al buffer de línea los correspondientes pixels decodificados. Este proceso continúa hasta que es llenado el buffer de línea con una línea completa para cada plano de color (uno para monocromático o 4 para 16 colores). Por último, la función fija los registros secuenciadores para el plano de color apropiado y transfiere una línea de datos a la memoria de despliegue para cada plano de color. Los datos son transferidos sólo para las líneas a ser desplegadas y sólo para aquella parte de la línea que será desplegada.

A continuación se dá el código C que despliega un archivo en formato .IMG por el método 1.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <dos.h>

#define sec_sal(registro, valor)  {outp(0x3C4, registro); outp(0x3C5, valor);}
#define graf_sal(registro, valor) {outp(0x3CE, registro); outp(0x3CF, valor);}
#define prox_caracter(ch, indice) {ch = arch_buf[indice];\
```

```
                                        if ((indice = (++indice) % 16384) == 0)\
                                          fread(arch_buf, 16384, 1, arch_IMG);}
#define prox_comp(ch, indice)     (ch =~ arch_buf[indice];\
                                        if ((indice = (++indice) % 16384) == 0)\
                                          fread(arch_buf, 16384, 1, arch_IMG);}

int restaura_pantalla(char nom_arch[]);
void fijaModo(int modo);
void fijaEGApaletas(char *col_paleta);

union REGS reg;

struct SREGS segregs;

long int  fireseg, fntoff, destseg, destoff;
int patron_tamano, xres, yres, modo, fin_x, fin_y;
unsigned char cabecera[16];
/*char paleta[16] = {0,3,5,1,20,2,4,7,56,59,61,57,62,58,60,63};*/
char paleta[16] = {0,3,5,1,6,2,4,8,7,11,13,9,14,10,12,15};

main(int argc, char *argv[])
{
  fijaModo(3);
  restaura_pantalla(argv[1]);
  getch();
  fijaModo(3);
}

/*************************************************************************

  fijaModo() = Fija el modo de video

*************************************************************************/

void fijaModo(int imodo)
{
  modo = imodo;
  reg.h.ah = 0;
  reg.h.al = modo;
  int86(0x10,&reg,&reg);
}

/*************************************************************************

  restaura_pantalla() = Lee y despliega un archivo .IMG

*************************************************************************/

int restaura_pantalla(char nom_arch[])
{
  FILE *arch_IMG;
  unsigned char ch,buffer[4][8192],arch_buf[16384],plano_sel[4] = {1,2,4,8};
  int i,j,k,col,renglon,ren_duplicados = 1,No_planos,plano,
     max_col,max_renglon, inicial_col = 0, inicial_renglon = 0;
  long int indice = 0;

  segread(&segregs);
```

```
fteseg = segregs.ds;
fntoff = (int) buffer;
destseg = 0xA000;
if((arch_IMG=fopen(nom_arch,"rb")) == NULL)
{
  printf("\nNo puedo Hallar %s.\n",nom_arch);
  return;
}
fread(cabecera,1,16,arch_IMG);
if(memcmp("\x00\x01\x00\x08\x00",cabecera,5))
{
  printf("\nFormato inapropiado en el archivo %s...\n",nom_arch);
  return;
}
patron_tamano = cabecera[7] + (cabecera[6] << 8);
No_planos = cabecera[5];
max_col = cabecera[13] + (cabecera[12] << 8);
max_renglon = cabecera[15] + (cabecera[14] << 8);
xres = 640;
if(max_renglon >= 480)
{
  modo = 18;
  yres = 480;
}
else
{
  modo = 16;
  yres = 350;
}
if(max_col > 8192)
{
  printf("\nResolucion no soportada...\n");
  return;
}
if((max_col<xres) || (max_renglon > yres))
{
  printf("\n Ancho de pantalla: %d",xres);
  printf("\n Ancho de imagen: %d",max_col);
  printf("\n Entre la columna inicial: ");
  scanf("%d",& inicial_col);
  printf("\n Alto de pantalla: %d",yres);
  printf("\n Alto de imagen: %d",max_renglon);
  printf("\n Entre la linea inicial: ");
  scanf("%d",& inicial_renglon);
}
fread(arch_buf,16384,1,arch_IMG);
fijaModo(modo);
fijaEGApaletas(paleta);
graf_sal(8, 0xFF);
indice = 0;
renglon = 0;
fin_x = min(xres/8,(max_col - inicial_col)/8);
fin_y = min( inicial_renglon + yres, max_renglon);
while(renglon < fin_y)
{
  col = 0;
  plano = 0;
```

```
ren_duplicados = 1;
while (plano < No_planos)
{
    prox_caracter(ch, indice);
    if (ch == 0x00)
    {
     j = patron_tamano;
     prox_caracter(ch,indice);
     if(ch == 0x00) /* Caso 00 00 FF NN. La próxima línea se repetirá */
     {
       prox_caracter(ch, indice); /* Se lee FF */
       prox_caracter(ren_duplicados, indice); /* Se lee NN */
     }
     else /* Caso 00 NN (ch es igual a NN). Corrida de un patron */
     {
       while (j--)
            prox_comp(buffer[plano][col++],indice); /* Se almacena el C1
            (complemento a 1) del próximo caracter en buffer[plano][col++] */
            k = ch - 1;
            while (k--)
            {
              memcpy(&buffer[plano][col],&buffer[plano][col-
              patron_tamano],patron_tamano); /* Se almacenan NN patrones
              en el buffer de línea */
              col += patron_tamano;
            }
     }
    }
    else
    {
      if(ch == 0x80)  /* Caso 80 NN. Cadenas de bits */
      {
            prox_caracter(i, indice); /* Se lee NN */
            while (i--)
              prox_comp(buffer[plano]
                    [col++],indice); /* Se almacena el C1 del próximo
                     caracter en buffer[plano][col++] */
      }
      else
      {
            if (ch > 0x80) /* Caso de corridas sólidas */
            /* ch > 0x80 implica corrida de pixels negros */
            {
              i = ch & 0x7F; /* Se deja en i la longitud de la corrida */
              while (i--)
                    buffer[plano][col++]  = 0x00;
            }
            else /* Corrida de pixels blancos */
            {
              i = ch & 0x7F; /* Se deja en i la longitud de la corrida */
              while (i--)
                    buffer[plano][col++] = 0xFF;
            }
      }
    }
    if (col >= max_col/8) /* Se termina con un plano y se pasa al siguiente */
    {
```

```
            col = 0;
            plano++;
          }
        }
        while(ren_duplicados -- > 0)
        {
          if (No_planos == 1)
          {
            sec_sal(2, 0xFF);
            if(renglon >= inicial_renglon)
            {
                destoff = (renglon - inicial_renglon) * 80L;
                movedata(fteseg,fntoff+( inicial_col/8),destseg,destoff, fin_x);
            }
          }
          else
          {
            for(i = 0; i < No_planos; i++)
            {
                sec_sal(2,plano_sel[i]);
                if(renglon >=  inicial_renglon)
                {
                  destoff = (renglon -  inicial_renglon) * 80L;
                  movedata( fteseg,
                        fntoff+( inicial_col/8)+8192*i,
                        destseg, destoff, fin_x);
                }
            }
          }
        }
        renglon++;
    }
    graf_sal(8,0xFF);
    graf_sal(5,0);
    fclose(arch_IMG);
  }

/*************************************************************************

  fijaEGApaletas() = Fija todos los registros de la paleta EGA y los
                     registros del borde

*************************************************************************/

void fijaEGApaletas(char *col_paleta)
{
    struct REGPACK regs;
    struct SREGS sregs;

    regs.r_ax = 0x1002;
    segread(&sregs);
    regs.r_es = sregs.ds;
    regs.r_dx = (int)col_paleta;
    intr(0x10,&regs);
}
```

PCX

De acuerdo al proceso descrito en la sección 5.1 podemos notar que al restaurar una pantalla se puede no asignar un color al mismo registro de color del que originalmente fue obtenido y que los registros de la paleta pueden no seleccionar los mismos registros de color. Sin embargo el resultado es el mismo, porque cada registro de paleta apunta a un registro de color que contiene la misma información de color contenida en la pantalla original.

Por otra parte, el programa que despliega los archivos PCX generados con el proceso descrito en la sección 5.1 es muy simple. En primer lugar verifica que 769 bytes antes del fin de archivo esté la bandera 0x0C que indica la presencia inmediata de la paleta. Si no esta dicha bandera se manda el mensaje de error apropiado; de lo contrario, se procede a leer la paleta. A continuación se leen de la cabecera las dimensiones del dispositivo de despliegue y se fija el modo gráfico apropiado (recordar que el modo gráfico queda definido por tales dimensiones y el número de colores). Enseguida se fija la paleta VGA a la que se leyó, y por último nos posicionamos en el lugar del archivo donde empiezan los datos codificados de la imagen.

Ya en este punto, la decodificación de los datos es simple: Se lee un byte y si tiene prendidos sus dos bits más significativos, se trata de una cuenta, caso en el cual se obtienen los 6 bits menos significativos y éstos nos darán el número de veces que se graficará el próximo byte. De lo contrario (el byte no tiene sus dos bits más significativos prendidos), el byte es graficado directamente.

A continuación se da el fragmento de código C correspondiente:

```
fseek(arch_pcx, -769L, SEEK_END);
ch = fgetc(arch_pcx);
if(ch != 0x0C)
{
   printf("\nEl archivo no está en modo 19.\n");
   fclose(arch_pcx);
   return(1);
}
for(i = 0; i < 256; i++)
   for(j = 0; j < 3; j++)
   {
      Paleta_elegida[i][j] = fgetc(arch_pcx);
      Paleta_elegida[i][j] = Paleta_elegida[i][j] / 4;
   }
fseek(arch_pcx, 12L, SEEK_SET);
fread(&xres, sizeof(int), 1, arch_pcx);
fread(&yres, sizeof(int), 1, arch_pcx);
if((xres == 320) && (yres == 200))
   fijaModo(0x13);
else
{
   if((xres == 640) && (yres == 480))
      fijaModo(0x67);
   else
```

```
    {
        printf("\nEl archivo no esta en modo 13h o 67h.\n");
        fclose(arch_pcx);
        return(1);
    }
}
fijaVGApaleta(&Paleta_elegida[0][0]);
fseek(arch_pcx, 128L, SEEK_SET);
for(k = 0; k < yres; k++)
{
    for(i = 0; i < xres; i++)
    {
        ch1 = fgetc(arch_pcx);
        if((ch1 & 0x0C) != 0x0C)
        {
            plot(i, k, ch1);
        }
        else
        {
            ch1 &= 0x3F;
            cuenta = ch1;
            ch = fgetc(arch_pcx);
            for(m = 0; m < cuenta; m++)
            {
                plot(i++, k, ch);
            }
            i--;
        }
    }
}
```

GIF

Debido a que la función dada en el capítulo anterior captura un archivo RGB a formato GIF guardándolo como una imagen de 320 * 200 pixels en 256 colores (modo 13h), la función que despliega los archivos así generados no necesita considerar otros modos de despliegue. Además, los archivos GIF generados aquí siempre cuentan con una tabla global de colores y no se dan tablas locales.

El código C siguiente ilustra el método de despliegue de un archivo .GIF en modo 13h:

```
void restaura_pantalla(char nom_arch[])
{
  char color;

  if ((arch_GIF = fopen(nom_arch,"rb")) == NULL)
  {
    printf("\nNo puedo Hallar %s.\n",nom_arch);
    return;
  }
  else
  {
    fread(&cabecera_gif,1,13,arch_GIF);
    if ((cabecera_gif.nombre[0] != 'G') ||
     (cabecera_gif.nombre[1] != 'I') ||
     (cabecera_gif.nombre[2] != 'F'))
    {
    printf("\n%s No es un archivo GIF valido .\n",nom_arch);
    fclose(arch_GIF);
    return;
    }
  }
  fijaModo(3);
  bandera_color = (cabecera_gif.empaquetado & 0x80) >> 7;
          /* Si M = 1, sigue de mapa de colores global */
  color_res = (cabecera_gif.empaquetado & 0x70) >> 4;
        /* cr + 1 = # bits de resolucion de color */
  tam_tab_col = (int)pow(2,(cabecera_gif.empaquetado & 0x07) + 1.0);
        /* 2^(pixel + 1) = tamaño de la tabla de colores */
        /* pixel + 1 = # bits/pixel en la imagen */
  if (bandera_color > 0)
    fread(tabla_de_colores,1,tam_tab_col * 3, arch_GIF);
  if (cabecera_gif.empaquetado == 0xF7)
    for (i = 0; i < 256; i++)
    {
    tabla_de_colores[i].red   >>= 2;
    tabla_de_colores[i].green >>= 2;
    tabla_de_colores[i].blue  >>= 2;
    }
  else
  {
    printf("\n%s no es un archivo en modo 19.\n",nom_arch);
    fclose(arch_GIF);
    return(1);
```

```
  }
  rangiones = 0;
  acabado = 0;
  while (!acabado)
  {
    ch = fgetc(arch_GIF);
    switch (ch)
    {
    case ';':    /* Fin de los Datos GIF */
        acabado = 1;
        break;
    case '!':    /* Bloque de extensión GIF */
                 /* leerlo y descartarlo    */
        fgetc(arch_GIF);
        i = fgetc(arch_GIF);
        fread(arch_buf,i,1,arch_GIF);
        if (fgetc(arch_GIF) != ',')
        break;
    case ',':    /* Leer el Descriptor de Imagen */
        fread(&descriptor_imagen,1,9,arch_GIF);
        ancho_de_linea = descriptor_imagen.ancho;
        altura_del_despliegue = descriptor_imagen.alto;
        bandera_de_tabla_global_de_col = (descriptor_imagen.empaquetado & 0x80) >> 7;
        interlace_bandera = (descriptor_imagen.empaquetado & 0x40) >> 6;
        if((ancho_de_linea == 320) && (altura_del_despliegue == 200))
        {
          modo = 19;
          xres = 320;
          fijaModo(modo);
          fijaVGApaleta(tabla_de_colores);
          decodificador(ancho_de_linea);
        }
        break;
    default:
        acabado = 1;
        break;
    }
  }
  fclose(arch_GIF);
}
```

En primer lugar, es leída la cabecera y descriptor de pantalla lógica en una estructura cabecera_gif (definida de manera idéntica que en el codificador), para a continuación verificar que en realidad se trata de un archivo GIF asegurándose de que sus 3 primeros caracteres forman la cadena "GIF". Después, utilizando el subcampo bandera_color (el cual el codificador dado aquí siempre fija a 1) del campo empaquetado se verifica la presencia de una tabla de colores global, caso en el que este presente es leída en el arreglo tabla_de_colores[] y (si se trata de un archivo en modo 13h) modificada para ser compatible con la tarjeta VGA.

Una vez hecho lo anterior, la función entra a una sentencia switch, la cual considera tres acciones a tomar dependiendo del siguiente caracter leído del archivo:

1) Si el caracter es un ";", significa que los datos GIF han llegado a su fin y termina el programa. Aquí es importante hacer notar que en la transmisión de imágenes, un flujo de datos GIF puede solo ser utilizado para proporcionar una paleta para los subsiguientes flujos GIF, por lo que es posible que los datos GIF terminen inmediatamente después de aparecer la tabla global de colores.

2) Si el caracter es un "!", el próximo bloque de datos es un bloque de extensión, el cual es saltado (en este programa no se procesan extensiónes) para a continuación leer el próximo caracter, y si éste no es una "," el archivo no contiene imagen alguna y por tanto termina la función.

3) Por último, si el primer caracter leído o el caracter leído después de un bloque de extensión es una "," es leído el descriptor de imagen y, si se trata de un archivo en modo 13h, es fijado el modo, fijada la paleta a la especificada y decodificada la imagen mediante la función decodificador() para ser desplegada.

La función decodificador() decodifica los datos en cadenas de pixels listos para ser desplegados. El código C correspondiente a decodificador() es:

```
void decodificador (int ancho)
{
  int codigo, fc = 0, viejo_codigo = 0, contador;
  int ch, tamano;
  int cad_indice = 0;
  int lin_indice = 0;

  tamano = fgetc(arch_GIF);
  tam_codigo = tamano + 1;
  tope = 1 << tam_codigo;
  terminador = 1 << tamano;
  fin_info = terminador + 1;
  entrada = nuevos_codigos = fin_info + 1;
  contador = ancho;
  bits_izquierda = 0;
  b1 = 0;
  bytes = 0;
  while ((ch = proximo_codigo()) != fin_info)
  {
   if (ch == terminador)
   {
     tam_codigo = tamano + 1;
     entrada = nuevos_codigos;
     tope = 1 << tam_codigo;
     ch = proximo_codigo();
     viejo_codigo = fc = ch;
     lin_de_despliegue[lin_indice++] = ch;
     contador--;
   }
   else
   {
     codigo = ch;
     if (codigo >= entrada)
```

```
    {
      codigo = viejo_codigo;
      Pila[cad_indice++] = fc;
    }
    while (codigo >= nuevos_codigos)
    {
      Pila[cad_indice++] = ultimo[codigo];
      codigo = liga[codigo];
    }
    Pila[cad_indice++] = codigo;
    if (entrada < tope)
    {
      fc = codigo;
      ultimo[entrada] = codigo;
      liga[entrada++] = viejo_codigo;
      viejo_codigo = ch;
    }
    if (entrada >= tope)
      if (tam_codigo < 12)
      {
        tope <<= 1;
        ++tam_codigo;
      }
    while (cad_indice > 0)
    {
      lin_de_despliegue[lin_indice++] =
        Pila[--cad_indice];
      if (--contador == 0)
      {
        sal_linea(lin_de_despliegue, ancho_de_linea);
        if (renglones >= altura_del_despliegue)
            return;
        lin_indice = 0;
        contador = ancho;
      }
    }
  }
  if (contador != ancho_de_linea)
    sal_linea(lin_de_despliegue, ancho_de_linea - contador);
}
```

La función empieza por leer el tamaño mínimo de un código LZW y a partir de ese valor se establecen los valores para tope (variable que establecerá el tope antes de incrementar la longitud de código), el código terminador y el código fin_info. La entrada en la tabla y el punto en el cual nuevos códigos pueden empezar es establecida una unidad más adelante del valor fin_info. La función entra entonces a un while que termina con la lectura del código fin_info. Si el código leído en una iteración del while es el código terminador, son reinicializados los parámetros de descompresión, y obtenido el valor del próximo código para ser salvado en el buffer de línea de despliegue. Si el código leído no fue un código terminador, primero se checa si no se está en un caso anormal donde un código es más largo que cualquier código en la tabla. Entonces se guarda el valor del último caracter dado por el código en la Pila. A continuación, el valor liga (dado por el código actual) es asignado a la variable código, se guarda el valor del último caracter en la Pila y se actualiza código,

hasta que se obtiene un valor para código menor que nuevos_códigos, indicando con esto que tenemos un código para un solo caracter (el cual es el primer caracter de la cadena). A continuación son guardadas las entradas para último y liga para esa nueva cadena en la tabla, incrementando el valor de entradas y, si es mayor que tope, incrementando el número de bits para un código. Por último, la función entra a un while que toma todos los caracteres de la cadena acumulados en la Pila y los deja en el buffer de línea de despliegue. Cuando un contador indica que ese buffer ha acumulado una línea completa, la línea es desplegada (por una función sal_linea) y el contador es actualizado.

La función sal_linea() que despliega una línea de longitud longitud_de_linea es:

```
void sal_linea(char *pixels, int longitud_de_linea)
{
  char far *direccion;

  for (i = 0; i < longitud_de_linea; i++)
  {
    direccion = (char far *) 0xA0000000L + xres * (long)renglones + (long)i;
    *direccion = pixels[i];
  }
  renglones++;
}
```

Por último, la función proximo_codigo() utilizada en el bucle principal de decodificador() tiene dos funciones:

1) Si arch_buf esta vacio (se terminó un paquete de bytes), lee el próximo caracter (el tamaño del siguiente paquete de bytes) del archivo y a continuación lee un bloque de ese tamaño en arch_buf.

2) Utiliza cuantos bits estén disponibles en un caracter previo mas cuantos bits sean necesarios del próximo caracter para obtener el próximo código de los datos empaquetados en el buffer.

```
int proximo_codigo()
{
   int bandera = 0;
   unsigned long int codigo;

   if (bits_izquierda == 0)
    bandera == 1;
   codigo = b1 >> (8 - bits_izquierda);
   while (tam_codigo > bits_izquierda)
   {
    if (bytes <= 0)
    {
      indice = 0;
      bytes = fgetc(arch_GIF);
      fread(arch_buf,bytes,1,arch_GIF);
    }
    b1 = arch_buf[indice++];
    if (bandera == 1)
    {
```

```
    codigo = b1 >> (8 - bits_izquierda);
    bandera == 0;
  }
  else
    codigo |= b1 << bits_izquierda;
  bits_izquierda += 8;
  --bytes;
 }
 bits_izquierda -= tam_codigo;
 codigo &= 0xFFF >> (12 - tam_codigo);
 return((int)(codigo));
}
```

CONCLUSIONES

Existe una amplia disparidad entre diferentes formatos de almacenamiento de imágenes, aunque cada formato contiene básicamente la misma información. Además de los descritos en este trabajo existen muchos otros formatos "coloquiales", cada uno con su propio conjunto de dificultades. Hay traductores de dominio público y privado que convierten un archivo de un formato a otro. Sin embargo, el número de traductores necesarios crecerá al cuadrado del número de formatos distintos ($n^2 - n$). Lo que es necesario en la "comunidad gráfica" es una forma intermedia de estándar común, tal como existe en otras áreas de publicidad. En este esquema es escrito un programa de traducción para cada forma variante a la forma intermedia, y viceversa. Así, solo se necesitarán 2n traductores para traducir de un formato a cualquier otro. El proyecto Chameleon en la Universidad del Estado de Ohio ha investigado ese proceso para documentos textuales y para archivos estándares de intercambio de gráficas [MOP89].

Por otra parte, en lo que respecta al emergente estándar JPEG de compresión de imágenes de tonos continuos, éste no es una panacea que resolverá la gran cantidad de problemas, los cuales deben ser atacados antes de que las imágenes digitales sean integradas por completo en todas las aplicaciones que finalmente se beneficiarán con ellas. Por ejemplo, si dos aplicaciones no pueden intercambiar imágenes comprimidas porque utilizan espacios de color incompatibles, tasas de aspecto de pixel, dimensiones, etc., entonces un metodo de compresión común no será útil.

Aunque el estándar JPEG proporciona una SINTAXIS DE INTERCAMBIO DE FORMATOS, la cual asegura que una imagen JPEG-comprimida puede ser intercambiada exitosamente entre diferentes ambientes de aplicación, un gran número de aplicaciones están "atascadas" debido a los costos de almacenamiento o transmisión, a la disyuntiva de cual metodo (no estándar) de compresión utilizar, o a que los codecs VLSI's son muy caros por sus bajos volúmenes. Para esas aplicaciones, el trabajo de evaluación técnica, pruebas, seleccion, validación y documentación que los miembros del comité JPEG han desarrollado, se espera que conduzca pronto a un estándar aprobado internacionalmente que resistirá las pruebas de calidad y tiempo conforme las diversas aplicaciones de imágenes lleguen a ser crecientemente implementadas en sistemas abiertos y redes de computación.

La última medida del éxito del comité será cuando las imágenes digitales JPEG-comprimidas lleguen a ser vistas y tomadas por concesión como justamente "otro tipo de datos", como lo son texto y gráficas hoy en día.

REFERENCIAS

[AMTIFF] Aldus Corporation Developer's Desk, Microsoft Corporation Windows Marketing Group, TIFF 5.0, *An Aldus/Microsoft Technical Memorandum*. 6/8/88

[AJ80] Arun N. Netravali and John O. Limb, "Picture Coding: A Review", *Proceedings of the IEEE*, Volumen 68, Número 3, Marzo de 1980, páginas 366-406.

[CDF86] Campbell, G., T. DeFanti, J. Frederiksen, S. Joyce, L. Leske, J. Lindberg, and D. Sandin, "Two Bit/Pixel Full Color Encoding", *Computer Graphics*, Volumen 20, Número 4, Agosto de 1986, páginas 215-223.

[Carl91] Carlson, W. E., "A Survey of Computer Graphics Image Encoding and Storage Formats", *Computer Graphics* 25(2): Abril de 1991, páginas 67-75.

[CIE86] Commission Internationale de l'Eclarage 1986, CIE *Colorimetry: Official Recommendations of the International Commission on Illumination*, Publication 15-2.

[Com90] CompuServe Incorporated, *Graphics Interchange Format*. Julio de 1990.

[Foley] Foley, Van Dam, Feiner, Hughes. *Computer Graphics: principles and practice*, Second Edition Addison Wesley, The Systems Programming Series.

[Gib85] Gibbons J. D. *Nonparametric Statistical Inference*. Segunda edición Marcel Dekker, New York 1985.

[GW87] González, R. C. and P. Wintz. *Digital Image Processing*. Addison Wesley, Reading, Mass., 1987.

[Graef89] Graef, G. L., "Graphics Formats", *Byte*: Septiembre de 1989, páginas 305-310.

[Hec82] Heckbert, Paul., "Color Image Quantization for Frame Buffer Display", *Computer Graphics* 16(3): Julio de 1982, páginas 297-307.

[Huf52] Huffman, D. A., "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the IRE*, Volumen 40, Número 9, Septiembre de 1952, páginas 1098-1101.

[ICC85] International Consultive Committee on Telegraph and Telephone (CCITT) Geneva, 1985, Volumen VII, Fascículo VII.3, *Terminal Equipment and Protocols for Telematic Services*. Páginas 16-31 y 40-48, respectivamente, para facsímile Grupo 3 y Grupo 4.

[LS71] Landau, H. J., and D. Slepian. "Some Computer Experiments in Picture Processing for Bandwidth Reduction" *Bell Systems Technical Journal*, 50:1525-1540, Mayo y Junio de 1971.

[KJ92] Kay, D. C. y John R. Levine, *Graphics File Formats*, Primera edición Windcrest/McGraw-Hill 1992.

[Loh84] Lohscheller, H., "A Subjectively Adapted Image Communication System", *IEEE Trans. Commun.* COM-32: Diciembre de 1984, páginas 1316-1322.

[Lyn73] Lynch, Michael F., "Compression of Bibliographic Files Using an Adaptation of Run-Length Coding", *Information Storage Retrieval*, Vol. 9: 1973, páginas 207-214.

[MOP89] Mamrak, S. A., C. S. O'Connell, and R. E. Parent, "The Automatic Generation of Translation Software for Graphic Objects", *IEEE Computer Graphics & Applications*, 9(6), Noviembre de 1989, páginas 34-42.

[Mark92] Mark, Nelson, *The Data Compression Book*, M & T Books 1992.

[MS90] Microsoft Corp., *Microsoft Windows Programmer's Reference*, Microsoft Press, 1990, ISBN 1-55615-309-0.

[Pom90] Pomerantz, Dave, "A Few Good Colors", *Computer Language*: Agosto de 1990, páginas 32-41.

[Reg81] Reghbati, H. K., "An Overview of Data Compression Techniques", *IEEE Computer*, 14(4): Abril de 1981, páginas 71-76.

[Rim92] Rimmer, Steve, *The Graphic File Toolkit*, Addison Wesley 1992.

[Stor92] Storer, J. A., *Image and Text Compression*, Kluwer Academic Publishers 1992.

[TGA91] Truevision Targa File Format Specification Version 2.0. *Technical Manual Version 2.2*. Enero de 1991. Indianapolis: Truevision, Inc.

[Wal91] Wallace, Gregory K., "The JPEG Still Picture Compression Standard", *Communications of the ACM*, 34(4): Abril de 1991, páginas 31-44.

[Wel84] Welch, T. A., "A Technique for High-Performance Data Compression", *IEEE Computer*, 17(6): Junio de 1984, páginas 8-19.

[Win72] Wintz, P. A. "Transform Picture Coding", *Proceedings of the IEEE*, Volumen 60, Número 7, Julio de 1972, páginas 809-820.

[WP71] Wilkins, L. C., and P. A. Wintz, "Bibliography on Data Compression, Picture Properties, and Picture Coding", *IEEE Transactions on Information Theory*, IT-17(2): Marzo de 1971, páginas 180-197.

[ZL77] Ziv, J. and A. Lempel. "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory*, IT-23(3): Mayo de 1977, páginas 337-343.

[ZL78] Ziv, J. and A. Lempel. "Compression of Individual Sequences via Variable-Rate Coding", *IEEE Transactions on Information Theory*, IT-24(5): Septiembre de 1978, páginas 530-536.

[Zsoft91] Zsoft Corporation. 1991. *Technical Reference Manual*.