UNIVERSIDAD NACIONAL AUTÓNOMA DE MEXICO

Escuela Nacional de Estudios Profesionales
"ARAGÓN"

TEORIA DE LA COMPRESION DE DATOS

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
INGENIERO EN COMPUTACIÓN
P R E S E N T A :

EDMUNDO RODRIGUEZ GARCIA

ASESOR: ING. SILVESTRE LOPEZ ABUNDIO

ENEP
ARAGON

SAN JUAN DE ARAGÓN, EDO. DE MEXICO 1995

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

TEORIA DE LA COMPRESION DE...

PROLOGO

El objetivo del presente trabajo es el de dar a conocer una herramienta, que básicamente reduce la longitud de cualquier archivo, permitiendo con ésto ocupar un menor espacio en cualquier unidad de almacenamiento donde se puede duplicar y en ocasiones triplicar su capacidad, además es muy utilizado en la transferencia de archivos via telefónica, ya que los archivos previamente comprimidos se transmitirán más rápido que uno no comprimido, ahorrando con ésto tiempo y dinero.

En el capítulo uno se verá la importancia de contar con la disponibilidad de espacio en una unidad de almacenamiento, así como a los dispositivos en donde se utiliza ésta herramienta. Básicamente muestra los conceptos primordiales que se utilizarán a lo largo del desarrollo de éste trabajo, esto es, conocer los elementos de las computadoras personales (PC's) para almacenar nuestra información vital.

En el capítulo dos se define el concepto de la compresión, su historia, los tipos de compresión existentes, así como las características que deben cumplir para mantener seguridad en los datos que se manejan.

En el capítulo tres, se comprenderá cómo funciona un compresor de datos por medio del pseudocódigo, los algoritmos y listados de programas. Obteniendo un cuadro comparativo de la compresión de éstos programas en diferentes archivos.

En el capítulo cuatro comprende como la seguridad de los compresores aún se encuentra en una fase de desarrollo, ya que ha logrado despertar inquietudes en aquellos usuarios que han tenido catastróficas pérdidas de información. La seguridad es un punto importante que de no tomarla en cuenta no se logrará la confianza plena en un producto novedoso y revolucionario.

En el capítulo cinco se encuentra la importancia que puede llegar a tener la compresión de datos, así como las ventajas y desvantajas al utilizarlo, como conclusión a éste trabajo.

TEORIA DE LA COMPRESION DE DATOS

INTRODUCCION

La primera vez que oí hablar de la compresión de datos, me era inconcebible la idea de que la información de archivos de texto, bases de datos y más aún de programas ejecutables se reducian para ocupar un menor espacio en nuestra unidad de almacenamiento ya sea en discos flexibles, duros y ópticos, en cintas magnéticas o cualquier otro medio que permitiera respaldar o guardar nuestra información.

Actualmente la compresión se ha vuelto algo indispensable para cualquier unidad de almacenamiento no sólo por el aumento de espacio que logra, sino por la necesidad que tienen los programas de aplicación ya sea Windows, Excel, Word y los mismos sistemas operativos en la utilización de dichos espacios para lograr un mejor desempeño.

Es por esto que el presente trabajo aborda todo lo relacionado a la teoría de la compresión de datos, presentando un panorama general de su historia y su funcionamiento; también encontraremos como diseñar un compresor, así como sus ejemplos escritos en lenguage C. Además se aborda un tema interesante sobre la seguridad de los programas existentes en el mercado, para finalizar con el futuro de la compresión y que se espera de él.

Ahora bien, la compresión, ya sea a través de un programa o de un circuito es compleja y sofisticada porque tiene que regirse por las normas del sistema operativo por que en ocasiones son programas independientes y se ejecutan en ese ambiente ya sea MS-DOS, PC-DOS, OS/2, UNIX u otro sistema operativo.

Los programas se basan en el trabajo de dos profesores israelíes : Abraham Lempel y Jacob Ziv, quienes básicamente localizan cadenas repetidas de caracteres en un archivo y las convierten en pequeños símbolos. Cuando se necesita expandir el archivo, se invierte el proceso y todo vuelve a su forma original. El algoritmo de compresión que ellos publicaron en 1977 se conoce ahora como el algoritmo Lempel-Ziv y es el origen de esta técnica.

1.- ALMACENAMIENTO DE LA INFORMACION

Las unidades de almacenamiento tales como discos duros, discos flexibles, cintas magnéticas y ahora con la nueva tecnología de los discos ópticos con capacidad de grabar, son indispensables para los respaldos, existiendo en el mercado una gran variedad de marcas, capacidades y de diferentes precios, Por mucho tiempo se había pensado solamente en los dispositivos eléctricos y/o mecánicos con mayor capacidad para satisfacer dichas necesidades de espacio, dejando atrás los programas que también logran esta función sin la necesidad de adquirir otra unidad de almacenamiento de mayor capacidad.

En este capítulo se verá la diferencia entre los dispositivos de almacenamiento existentes en el mercado y los programas compresores, así como la estructura de los discos duros que nos darán una idea para entender el funcionamiento de los programas compresores.

Si conocemos la estructura de los discos y su funcionamiento, entenderemos cómo y en que áreas de los discos es posible almacenar datos y cómo los programas de compresión se basan en éstos para la optimización del espacio del disco.

1.1 PORQUE SE ALMACENAN LOS DATOS

La principal razón por lo que se almacena la información ya sea en medios magnéticos, ópticos y/o eléctricos, es la de mantener la información disponible por un periodo corto o largo de tiempo.

En un principio, sólo se contaba con el dispositivo eléctrico llamado RAM (Memoria de Acceso Aleatorio), que es la memoria convencional con que toda computadora cuenta; su función primordial es la de apoyar la ejecución de cualquier aplicación. Este tipo de almacenamiento puede mantener la información en forma temporal ya que depende del suministro de energía y la información contenida se "pierde" en caso de interrumpirla. Por lo que para evitar la pérdida de información se desarrollaron los sistemas de respaldo, en los cuales se lograba almacenarla por el tiempo que fuese necesario y disponer de éstos en cualquier momento.

Pensado en dispositivos externos para mantener la información, se habían desarrollado las cintas magnéticas parecidos a las cintas convencionales analógicas, pero con la desventaja de que el acceso es secuencial, lo que implica leer la cinta desde el principio para localizar un archivo deseado, (en ocasiones si el archivo se localiza en el principio de éste concluye la búsqueda; en caso contrario tendrá que leer toda la cinta hasta encontrarlo). Por ésta razón se abocaron a los discos flexibles, y luego a los discos duros. Pero ahora con la tecnología del discos ópticos se obtiene la ventaja de capacidad ofrecidas por las cintas magnéticas, y la ventaja del acceso directo ofrecido por los discos magnéticos, con la desventaja de tener una velocidad menor al disco duro.

Aún con toda la tecnología disponible las operaciones de lectura y escritura de archivos en cualquier medio magnético puede sufrir pérdida de información de manera accidental ya que aún no se controlan todas las variables para lograr una seguridad confiable al 100%, por lo que se sigue desarrollando nuevos medios más seguros y/o modificando los actuales para aumentar su confiabilidad.

1.2 IMPORTANCIA DE LA DISPONIBILIDAD DE ESPACIO EN UNA UNIDAD DE ALMACENAMIENTO

La importancia del espacio en nuestra unidad es de vital importancia, ya que siempre estaremos utilizando este medio para almacenar nuestra información. Cuando menos lo esperemos los archivos generados por las aplicaciones de los sistemas crecerán sin control a menos que se cuente con una buena administración de espacio. La compresión de datos ofrece la duplicación y en ocasiones triplican la capacidad de almacenamiento, comprimiendo todo un disco o parte de él. Esto no implica que se olvide de administrar los espacios ya que sólo es una herramienta más para aprovechar al máximo el espacio disponible.

Ahora bien, la compresión es aprovechado en los BBS (Boletín Electrrónico), donde cualquier usuario que tenga acceso a ellos y cuenten con un modem (modulador y demodulador), podrá transmitir o recibir información, directamente por vía telefónica desde o hacia cualquier parte del mundo; comprimiendo previamente sus archivos para ocupar el menor espacio posible se reducirá así el tiempo de transmisión o recepción.

1.3 DISPOSITIVOS DE ALMACENAMIENTO

Existen diversos dispositivos de almacenamiento, como son:
- Las cintas magnéticas
- Discos magnéticos (Duros - Flexibles)
- Discos ópticos
- Discos ópticos magnérticos

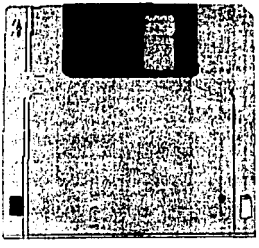
Las primeras formas de almacenar la información se lograba hacer en cintas magnéticas que era muy semejante a las grabaciones de cinta comerciales, o sea, en forma de pulsos eléctricos. Actualmente los dispositivos de resguardo estan diseñados para reconocer la información de la computadora, ya que ésta sólo reconoce la información como números binarios (ceros y unos).

Este sistema de almacenamiento es muy confiable y de bajo costo, por lo que esta al alcance de cualquier usuario, pero por su lentitud al grabar y recuperar datos sólo es utilizado principalmente para las copias de seguridad o de respaldo, ya que son medios de acceso secuencial, lo que significa que para buscar un programa o un dato que se encuentra almacenado físicamente al final de la cinta, se debe adelantar toda la cinta para encontrarlo y accesarlo.

1.4 CONOCIENDO LOS DISCOS DE ALMACENAMIENTO

Los discos magnéticos es un medio de almacenamiento informático. Como tal, requiere de un conjunto de normas que permitan la lectura y escritura de la información, de manera organizada, eficiente y estandarizada.

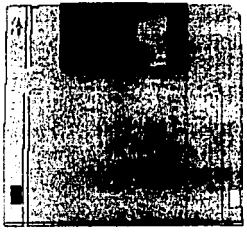
El mas usual de los medios de almacenamiento masivo es el disco duro y el flexible (también llamado "floppy" o "diskette"), que se asemeja a un disco fonográfico, pero que en lugar de surcos dispone de pistas y sectores magnéticos que permiten la organización lógica de la información. Este dispositivo fue diseñado por el inventor japonés Yoshiro Nakamats y, aunque originalmente se trato de un disco con un diámetro de 8 pulgadas, actualmente es un formato que ha caído en desuso. Concretamente, hoy en día existen dos tipos de discos flexibles comerciales, compatibles con IBM: Los de 5 1/4 pulgadas de diámetro, disponibles en versiones de 360 Kbytes y 1.2 Mbytes de capacidad; y los de 3 1/2 pulgadas de diámetro, disponibles en versiones de 720 Kbytes y 1.44 Mbytes. Recientemente están entrando al mercado los discos flexibles con capacidad de 2.88 Mbytes, los cuales vienen contenidos en una cubierta de plástico rígido y sellada por una lámina metálica; estos discos fueron diseñados en los laboratorios de Sony Corp. También se ha puesto a la venta discos flexibles de 3 1/2 pulgadas que trabajan auxiliados por un rayo láser para la localización de los sectores, tecnología que se conoce como "floptical disk" o discos ópticos-magnéticos, y con la cual se permite almacenar hasta 21 Mbytes de información en un sólo disco, siendo compatibles con las unidades de 720 KB y 1.44 MB.



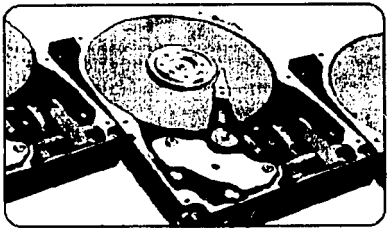
ESTRUCTURA FISICA DE UN DISKETTE DE 3.5"

ESTRUCTURA FISICA DE UN CARTUCHO DE 250 MB.

FIG 1 1 UNIDADES DE DISCO FLEXIBLE Y CINTA



ESTRUCTURA FISICA DE UN DISCO OPTICO 21 MB



ESTRUCTURA FISICA DE UN DISCO DURO DE 5.25"

FIG 1 2 UNIDADES DE DISCO OPTICO Y DISCO DURO

1.4.1 ESTRUCTURA FISICA DE UN DISCO MAGNETICO

Un disco magnético es un dispositivo de memoria externa o secundaria, basada en la escritura/lectura de sectores y pistas sobre una superficie móvil magnetizada. Está fabricado con una lámina de plástico recubierta de óxido magnético por ambos lados, y generalmente gira a una velocidad de 300 revoluciones por minuto, dependiendo del fabricante de dicha unidad.

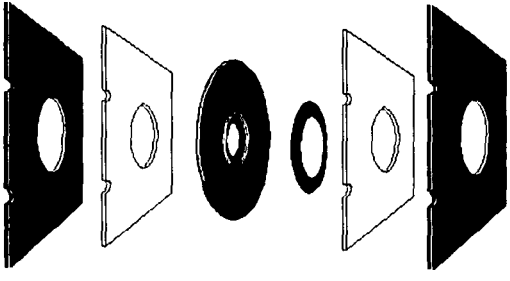


FIG 1.3 ESTRUCTURA INTERNA DE UN DISCO FLEXIBLE DE 5 1/4

La escritura y lectura de la información queda expresada en formato binario, y está a cargo de una cabeza de lectura/escritura. Lo anterior origina canales magnéticos que contiene codificada la información en código binario.

1.4.2 ¿QUE ES INICIALIZAR UN DISCO?

Recurriendo a una analogía que todos conocemos, podemos afirmar que un disco magnético nuevo es como un cuaderno cuyas páginas no disponen de renglones, o como un campo sin arar, por lo que no puede ser usado en esas condiciones: hay que INICIALIZARLO.

Para que el CPU pueda almacenar y accesar rápidamente la información en el disco magnético, se precisa de un sistema de direcciones (algo así como un sistema de coordenadas que identifiquen puntos en la superficie del disco); así, un disco debe ser preparado magnéticamente para que pueda almacenar lógicamente y de manera organizada la información.

A la acción de "dibujar" magnéticamente en cada cara del disco una serie de circunferencias o cilindros concéntricos, parecidos a los surcos de un disco musical, se le denomina "inicializar un disco o dar formato".

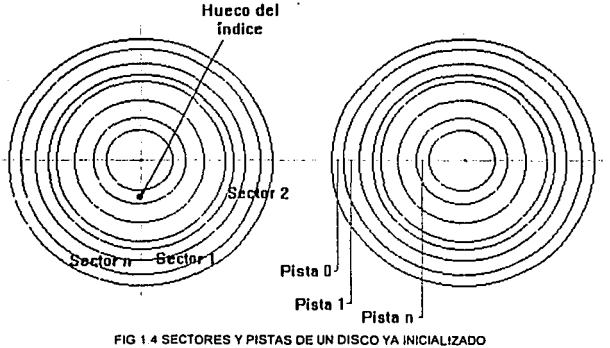


FIG 1.4 SECTORES Y PISTAS DE UN DISCO YA INICIALIZADO

A su vez, la inicialización se divide en dos etapas:

1) A lo que se le ha denominado "Formateo físico o de bajo nivel", que consiste en el "dibujo" magnético en cada cara del disco de pistas y sectores, algo parecido a la acción de pasar un tractor por un terreno virgen, para preparar las calles y avenidas que habrán de servir de base a la urbanización de un fraccionamiento.

2) La segunda etapa se ha llamado "Formateo lógico o de alto nivel", el cual se procesa inmediatamente al anterior, y consiste en dividir la capacidad de almacenamiento del disco en porciones que puedan ser manejadas por el sistema operativo, algo parecido a la definición del tipo de construcción de las casas y la asignación de la nomenclatura correspondiente por manzanas y lotes, según las normas del municipio, para el ejemplo del fraccionamiento.

Tanto el formateo físico como el lógico, en el caso de los discos magnéticos, dependen del sistema operativo en el que serán utilizados.

1.4.2.1 FORMATO FISICO O DE BAJO NIVEL

Durante el formato físico se inicializa el disco, para lo cual un par de cabezas de lectura/escritura, encontradas respectivamente con las dos superficies, "dibujan" con marcas magnéticas cierta cantidad de pistas o tracks. Por ejemplo, un disco normal de doble cara para una capacidad de 360 Kbytes, se da formato para 40 pistas numeradas de la 0 a la 39, cada una dividida en 9 sectores numerados del 1 al 9, cada uno de los cuales a su vez dispone de una capacidad de almacenamiento de 512 bytes (norma estándar IBM) se tiene entonces 40 cilindros (1) completos y una capacidad de almacenamiento de 368,640 bytes, que para abreviar simplemente se denominan 360 Kb. (2). (Para el Sistema Operativo DOS).

(1) Se llama cilindro al conjunto de pistas situadas perpendicularmente en la misma posición. En el caso del disco flexible solamente hay dos caras, por lo que cada cilindro está formado por dos pistas, una colocada en el lado 0 y la otra en el lado 1. En el caso del disco duro la existencia de varios platos o discos unidos entre sí, dará como resultado que cada cilindro se forme por las dos pistas de cada uno de los platos.

(2) 512 bytes por cada sector de cilindro por 2 caras es igual a 1,024 bytes, por 9 sectores por cilindro es igual a 9,216 bytes, por 40 cilindros es igual a 368,460 bytes totales en ambas caras del disco flexible.

Naturalmente que si el disco es de otra densidad o de otro diámetro, la capacidad de grabación de pistas y sectores también será otra. En la siguiente tabla se muestra cómo se conforma la capacidad de almacenamiento de los discos flexibles de 5 1/4 y 3 1/2 pulgadas en su densidad mas popular.

	DISCOS			
	5 1/4		3 1/2	
	360	1.2	720	1.44
Caras	2	2	2	2
Pistas, cilindros	40	80	80	80
Sectores por pista	9	15	9	18
Capacidad de almacenamiento en bytes	368,640	1,228,800	737,280	1,474,560

Tabla 1.1 Comparación entre los diferentes tipos de discos flexibles del sistema operativo DOS

Además de la delimitación del disco en sectores y pistas, es preciso disponer de un sistema de direcciones que permita almacenar y accesar de manera ordenada y lógica la información; es necesario, entonces, numerar los elementos que componen el disco (cilindros, caras, pistas, sectores). Hay dos modos de direccionamiento que durante el formato físico se determinan: la numeración física y la numeración lógica.

En el caso de la numeración física, la tarjeta controladora del disco y las rutinas de la memoria ROM BIOS especifican el sistema de direccionamiento en base a tres coordenadas: cara, pista (cilindro) y sector, por ejemplo: lado 0, cilindro 14, sector 16. Sin embargo, para el sistema operativo es mejor considerar los datos como si se trataran de una larga trenza de unidades secuenciales, por lo que numera los sectores de la misma pista y cara; luego se numeran los sectores de la misma pista pero situadas en la cara contraria, para continuar con el segundo sector de la cara 0, pista 1, etc. Esta es la numeración lógica. Con este método se minimizan los movimientos de las cabezas y, por lo tanto, se optimiza el tiempo de acceso cuando se intentan leer sectores situados de forma ordenada.

Llegando a este punto, cabe tener presente que la unidad controladora del disco flexible reconoce la posición del primer sector de cada pista mediante un pequeño orificio de indexación, ubicado cerca del centro del disco flexible (tipo 5 1/4"), el DOS también reconoce la posición de éste por medio del orificio. Para ello se proyecta un haz de luz sobre la ventana del orificio índice, el cual sólo podrá pasar en el momento en que coincida con dicha ventana; cada vez que ello ocurre, durante la rotación, se puede ir determinando la posición del disco.

Ahora bien, un sector es la unidad mínima que se puede leer o escribir en el disco; para ello, el DOS debe llevar un control del estado de cada uno (posición del sector, qué contiene, si está libre o físicamente dañado), lo que resulta muy costoso en cuanto a espacio de almacenamiento del propio disco, además de complicado y lento, dado que 512 bytes aún es una cantidad muy pequeña de información. Por lo tanto, el DOS agrupa un cierto número de sectores contiguos, llamados "clusters" (racimos), y los trata como un sólo paquete de información en el disco. Tenemos entonces un tercer método de direccionamiento de la información: mediante la numeración de clusters.

1.4.2.2 FORMATO LOGICO O DE ALTO NIVEL

Una vez que se ha creado las pistas y sectores de un disco, se precisa de un método que permita manejar eficientemente la información y llevar un recuento de su estado de organización. Este método se define durante el formato lógico.

En principio, tenga presente que en esta etapa el DOS organiza la información en clusters, como unidad mínima de almacenamiento. Naturalmente, los clusters no son una propiedad inherente al formato físico del disco, sino un método del sistema operativo para optimizar el manejo de los datos, por lo que el número de sectores (cada uno de 512 bytes) contenidos en cada clusters varía en función del tipo de disco magnético.

Tipo de disco	Tamaño del Clusters
5 1/4 pulgadas	
360 Kbytes	2 sectores
1.2 Mbytes	1 sector
3 1/2 pulgadas	
720 kbytes	2 sectores
1.44 Mbytes	1 sector

Tabla 1.2 Clusters para los diferentes tipos de discos flexibles comerciales.

El formato lógico consiste en la creación de un conjunto de índices que señalan en todo momento el estado del disco (qué zonas estan ocupadas, sectores defectuosos, cuánto espacio disponible hay y en dónde) y que permiten direccionar la información. Para ello el disco es dividido en dos zonas:

1) Area del sistema, dividida a su vez en: a) sector de arranque (boot sector), b) FAT (File Allocation Table: Tabla de localización de Archivos) y c) directorio raíz (root directory).

2) Zona de datos.

1.4.3 EL SECTOR DE ARRANQUE

Cuando se acciona el interruptor de encendido de la computadora, los programas de arranque del BIOS le indica leer el sector de arranque del disco, para comenzar a cargar en la RAM el sistema operativo. La computadora primero busca estas rutinas en el disco de la unidad A, y si no las encuentra, por estar vacía, pasa enseguida a buscarlas en el disco duro.

Si el usuario inserta un disco flexible auto-arrancable (es decir, que contienen los tres archivos básicos del sistema operativo), al encender la máquina podrá cargar en RAM las rutinas que le permitan trabajar; pero si el disco flexible no es auto-arrancable, se emitirá un mensaje de error tipo "NO ES DISCO SISTEMA O ERROR EN EL DISCO, REEMPLACE Y PRESIONE ALGUNA TECLA PARA CONTINUAR" (tal vez en inglés, dependiendo de la versión del DOS). Pues bien, tanto el trabajo de cargar el sistema operativo (o emitir el mensaje de error en caso dado) como el de informar al DOS de las características del disco, están a cargo del sector de arranque, para lo cual debe de disponer de dos partes: 1) Un programa muy corto de arranque (llamado IPL, por las siglas de Initial Program Loading), y 2) una lista de características clave del disco (número de bytes por sector, número total de sectores por disco, cantidad de sectores por pista, número de cabezas de lectura/escritura, etc.).

Debido a la importancia de estos datos, todos los discos flexibles del DOS, sean o no auto-arrancables, deben disponer de un sector de arranque, el cual siempre se ubica, en el caso de este tipo de discos, en el primer sector de la primera pista de la primera cara: sector 1, pista 0, cara 0.

Reservado	(3 bytes)
Nombre del fabricante y versión	(8 bytes)
Bytes por sector	(2 bytes)
Sectores por Cluster	(1 byte)
Sectores reservados, comenzando en 0	(2 bytes)
Número de copias de la Fat	(1 byte)
Número de entradas del directorio raíz	(2 bytes)
Sectores totales en el volumen lógico	(2 bytes)
Byte de formato del disco	
Número de sectores por FAT	(2 bytes)
Sectores por pista	(2 bytes)
Número de cabezas	(2 bytes)
Número de sectores ocultos	(2 bytes)
Rutina de arranque (Boot)	

Tabla 1.3 Mapa por bytes del sector de arranque de un disco del sistema operativo

1.4.4 LA FAT

La tabla de localización de archivos, FAT, consiste en un área que registra tanto el estado general de todos los clusters como el direccionamiento de los archivos, según la cadena de clusters que ocupan.

En un disco magnético un archivo rara vez queda grabado en un bloque de sectores contiguos (a menos que esté recién inicializado), como ocurre, digamos, con las canciones de un disco de audio, las cuales quedan grabadas completamente una a continuación de otra. Esto es así por que el DOS tiende a distribuir los archivos por toda la superficie del disco, aprovechando los espacios libres que van quedando por otros archivos que se han borrado o modificado. En consecuencia, los archivos tienden a quedar fragmentados, por lo que precisan de un método de encadenamiento lógico que permita seguirles la huella.

Justamente, para conocer tanto el estado de cada cluster (defectuoso, libre, ocupado, etc.) como su contenido y la cadena de información que permite seguirle el rastro a cada archivo, el DOS construye un mapa o tabla llamada FAT. En la FAT se asigna una entrada para cada cluster que contienen la información respectiva, cuyo número de identificación corresponde con el número de dicho cluster; por ejemplo, el cluster número 175 del disco esta asociado con la entrada número 175 de la tabla FAT.

Para entender mejor lo anterior, supongamos que un archivo llamado "TEXTO.DOC", cuya entrada en el directorio correspondiente dice que comienza con el cluster 85. El DOS lee los sectores de este cluster; luego consulta la posición 85 en la FAT, para averiguar cuál es el número del siguiente cluster, digamos que el 110. Enseguida el DOS lee la siguiente porción del archivo en el cluster 110 y consulta la posición 110 de la FAT, para encontrar el número del siguiente cluster en que continúa el archivo. Y así sucesivamente hasta que encuentra un código en particular (EOF=End Of File) que le indica que el archivo ha concluido.

Según el contenido de cada entrada en notación hexadecimal, se puede tener la información que se muestra a continuación:

ENTRADA EN LA FAT	SIGNIFICADO
0000	Cluster disponible
0002-FFEF	Cluster utilizado por un archivo. El contenido de esta entrada apunta al siguiente cluster del archivo.
FFF0-FFF6	Cluster reservado
FFF7	Cluster defectuoso
FFF8-FFFF	Ultimo cluster de un archivo

Tabla 1.4 Direcciones de la tabla FAT

Por otra parte, puesto que la tabla FAT contiene los índices de los archivos y sin ella la información sería muy difícil de recuperar, se graba de manera adjunta una copia como protección contra una pérdida de los datos, en caso de daño de algunos de los sectores de la FAT original. Y por último, el número de sectores que ocupa la FAT depende del tipo y la capacidad del disco, información que viene almacenada en el primer byte de la FAT.

1.4.5 EL DIRECTORIO RAIZ

Este es el índice que el DOS crea en la tercera y última parte del área del sistema durante el formato lógico. Tanto su tamaño como su posición quedan fijados durante esta etapa y no puede alterarse posteriormente.

La cantidad de sectores que ocupa el directorio raíz es distinta entre un disco de baja densidad y uno de alta; por ejemplo, en uno de 360 Kb de 5 1/4 pulgadas se ocupan 7 sectores, mientras que en uno de 1.2 Mb del mismo diámetro se usan 14.

El directorio raíz dispone de un cierto número fijo de entradas, en donde se recoge la información esencial de cada archivo a almacenar en el disco, con lo que la cantidad de archivos posibles a almacenar queda limitada por ese número de entradas. En los discos de 360 Kb y 720 Kb el número de entradas máxima es de 112, mientras que en los de 1.2 Kb y 1.44 Mb es de 224.

En la práctica, el directorio raíz se puede definir como el directorio de primer nivel en la estructura o árbol de archivos del disco, y se identifica con la diagonal inversa (\), también llamada "backslash".

Veamos ahora la siguiente tabla, la que nos muestra cuál es la información esencial de cada archivo que se registra en cada entrada del directorio raíz:

CAMPO	LONGITUD
Nombre	8 bytes
Extensión	3 bytes
Atributos	1 byte
Reservado	10 bytes
Hora	2 bytes
Fecha	2 bytes
Primer cluster del archivo	2 bytes
Tamaño	4 bytes

Tabla 1.5 Longitud en bytes sobre las características de un archivo en la tabla FAT

NOMBRE Y EXTENSION del archivo es una cadena de hasta 8 y 3 caracteres ASCII, respectivamente. El punto de separación entre ambos no queda almacenado.

El primer caracter de esta cadena tiene una importancia especial para el DOS. Por ejemplo, cuando se borra un archivo realmente lo único que se borra es el primer caracter, que se sustituye con un símbolo σ (sigma); también se borran sus entradas correspondientes en la FAT. Esto permite recuperar posteriormente tal archivo por medio de algunas técnicas, siempre que la cadena de clusters en que se aloja no haya sido ocupada por otro archivo. (El sistema operativo MS-DOS ver 5.00 se ha incorporado el comando UNDELETE, expresamente para estos casos).

En el campo de ATRIBUTOS se especifica la información que indica al DOS de las características del archivo correspondiente, como son: si es de sólo lectura, si es oculto, si es archivo del sistema, etiqueta de volúmen, si es subdirectorio o archivo.

Los 10 bytes que ocupa el "campo reservado" quedan para posibles usos futuros del DOS. Los dos siguientes campos señalan la HORA Y FECHA en que fue creado o sufrió la última modificación el archivo correspondiente.

El campo siguiente indica el primer cluster que ocupa ese archivo, es decir, actúa como punto de partida de la cadena de localización de la FAT. A partir de esta información el DOS puede seguir el rastro del archivo, leyendo las correspondientes entradas de la FAT.

El último campo contiene la información de cuántos bytes ocupa ese archivo.

1.4.6 ZONA DE DATOS

La zona de datos es la parte donde finalmente van a ser almacenados los archivos y programas del usuario, y es la parte que sigue a continuación del área del sistema, prolongándose hasta el final del disco, hacia el centro.

El DOS organiza la información en la zona de datos por clusters numerados secuencialmente, iniciando en todos los tipos de discos magnéticos por el cluster número 2; en consecuencia, el último cluster tendrá asignado un número que es mayor en 1 al realmente existente, como espacio disponible para almacenamiento en el disco.

1.5 FACTOR DE INTERCALACION O INTERFOLIACION

Las altas velocidades a las que giran los discos (3600 rpm en el caso de los discos duros) no permite que el sistema operativo DOS pueda leer la información en forma continua, ya que después de leer un sector y ubicar la información en memoria, cuando está listo opara leer el siguiente sector, éste puede ya haber pasado por debajo de la cabeza lectora y el DOS necesita esperar a que se produzca un giro completo del disco para leer el siguiente sector.

Para evitar ésta pérdida de tiempo y optimizar los tiempos de acceso (lectura escritura), los dicos flexibles o duros, se preparan desde su lugar de fabricación para que puedan grabar o leer la información con un factor de intercalación, que permite grabar o leer un sector y dejar pasar un número x de sectores, esperando el sector apropiado para grabar o leer el siguiente sector, y así, consecutivamente.

Una razón de intercalación de 4 a 1 significa que el sector con el siguiente número está físicamente a 4 sectores de distancia del anterior. Por ejemplo la secuencia de sectores en una pista de 4 a 1 sería: 0,13,9,5,1,14,10,6,2,15, 11,7,3,16,12,8,4. El resultado es que se lee 4 sectores por revolución en vez de sólo uno. Una razón de intercalación de 4 a 1 no es la ideal para todos los sistemas. La intercalación correcta para un sistema específico depende de forma primordial del controlador y de la velocidad del sistema.

Los factores de intercalación por omisión para sistema XT son de 6 a 1 y para AT de 3 a 1. Estos no son los factores óptimos. A menudo es posible mejorar la velocidad de transferencia de datos en un 10 o 20 por ciento si se reduce la intercalación a 5 a 1 o 2 a 1, respectivamente.

Información grabada en disco con factor de intercalación 4:1

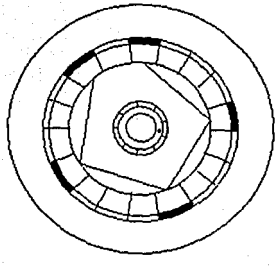


FIG 1.5 DIAGRAMA ESQUEMATICO DEL PROCESO DE INTERFOLIACION QUE BUSCA LOS DATOS DE CUATRO EN CUATRO SECTORES POR LO QUE SE LE DENOMINA INTERFOLIACION DE 4:1

2. COMPRESIÓN

La compresión de datos es un tema interesante, por lo que considero importante abarcar la definición del tema así como dar una idea general de su funcionamiento, no sin antes mencionar la historia de uno de los programas más populares de SHAREWARE.

2.1 DEFINICION

Pues bien, la COMPRESION, como su nombre lo indica, consiste en comprimir o reducir de tamaño algún objeto ya sea físico o lógico. En el área de la computación ésto puede causar confusiones ya que un archivo ésta lógicamente limitado y conforme a las normas del sistema operativo se pensaría que no sería posible reducirlo y aún menos utilizarlo; pues bien, con éste proceso es posible lograrlo, pero aún así no hay posibilidad de leerlo y hacer uso de él sin que exista su contraparte llamado EXPANSION que se encarga de invertir el proceso y dejar la información integra como antes de utilizar la compresión.

Las aplicaciones saltan a la vista, y solo un claro ejemplo son las unidades de almacenamiento que cuentan con esta herramienta y que pueden duplicar virtualmente su espacio, y ésto se logra reduciendo la información existente. Anteriormente, al no contar con el valioso espacio para nuestras aplicaciones, el pensamiento que surgía era adquirir una unidad de almacenamiento de mayor capacidad ya sea en disco duro o en cintas. Por suerte, el cambio ha llegado y con esto, se ha optado por adquirir éstos productos. Otro claro ejemplo, es la utilidad en la transmisión de archivos vía telefónica, en el que para enviar un archivo en el menor tiempo posible, es necesario que la longitud de éste sea significativamente pequeño, por lo que se ha optado por comprimirlos previamente facilitando la transmisión, en algunos casos el modem u protocolos utilizados ya emplean los algoritmos adaptados a un circuito.

2.2 EN QUE SE BASA LA COMPRESIÓN

La compresión, consiste en el reemplazo de cadenas repetidas de caracteres por un código específico ASCII, conservando la posición y longitud del caracter a representar; y el usar un código para representar varias cadenas de caracteres es el secreto de las rutinas de compresión.

Ahora bien, las hojas de cálculo, los archivos de texto, bases de datos y muchos programas ejecutables contienen en su gran mayoría cadenas que se repiten y que se pueden reducir. Un claro ejemplo son aquellos archivos de texto o archivos de documentos donde el algoritmo podrá representar a los tabuladores, márgenes, retornos de carro o los justificadores de textos con un solo caracter que se repiten innumerables veces.

En fin, la compresión es posible siempre y cuando existan en los archivos a comprimir cadenas repetidas, esto no es preocupante ya que en la mayoría de los archivos, resultantes de aplicaciones o archivos ejecutables se encontrarán dichas cadenas. Hay que notar que mientras mas grande es el archivo las posibilidades de encontrar cadenas repetidas se incrementa, y el archivo que contendrá ésta información se reducirá más con respecto al original.

Existen varios caminos para lograr ocupar el menor espacio por archivo, lo importante es lograr su integridad y rapidez, y algunos métodos simples podrían verse como:

a) Sustituir los caracteres y espacios que contiene un archivo por un caracter especial, éste es el método más utilizado.

b) Otra forma de compresión es la codificación por longitud de caracter, en un simple ejemplo se vería así : teniendo los caracteres a codificar como: "AAABBBBAACCCC" el codificador lo traduciría como "3A4B2A4C" lo que significa que existen tres "A", cuatro "B", dos "A" y por último cuatro "C"; éste método es el más común en la compresión de gráficas porque es muy frecuente encontrar caracteres iguales consecutivamente.

Existen en el mercado otro tipo de compresión con la característica de comprimir y descomprimir al momento de accesar a los archivos. A éstos se les ha denominado compresión en tiempo real, es decir, el programa decodificará la información en el momento de la lectura y codificará al momento de grabar.

Para comprender esto, es necesario comprender la disposición física de los archivos así como el dispositivo a utilizar, en nuestro caso nos enfocaremos al disco duro. Cuando el disco queda preparado para aceptar información, en donde el sistema operativo pueda usarla lógicamente, la forma dispuesta es la de pedazos de longitud fija llamado sectores. El formato fija el tamaño de los sectores como múltiplo de 512 bytes. El proceso de formato también agrupa los sectores en cluster o racimos. Cada cluster contiene un número fijo de sectores y esto se convierte en la unidad básica de almacenamiento de un disco. El tamaño del disco determina el número de cluster's que contiene, y este rígido sistema deja mucho espacio sin aprovechar.

Aunque un disco almacena la información en espacios de longitud fija, el archivo es una gran cadena que varía dependiendo de la información. Esto es, un disco duro tiene un formato con cluster's de cuatro sectores de 512 bytes, lo que significa que cada cluster tiene 2,048 bytes. En este formato, un archivo de 12,536 bytes, más o menos el largo de un documento de 2,100 palabras, ocuparía seis cluster's completos y 248 bytes del próximo cluster disponible cuando se guarde en el disco. Así con este sistema se deja de usar 1,800 bytes de un cluster y quedan perdidos para siempre como espacio de almacenamiento, ya que el sistema operativo guarda los archivos al principio de un cluster vacío y no al final de uno parcialmente lleno. En otras palabras, el sistema operativo ignora cientos de miles de bytes de espacio libre de disco que los productos de compresión usan sin problemas.

El hardware y software que utilice la compresión en tiempo real también utilizan manejadores de dispositivos para reemplazar la interfaz del disco duro o para decirle al sistema operativo que se ha añadido otro disco al sistema. Mientras el manejador de dispositivo se comporte de acuerdo con las reglas de los dispositivos del sistema operativo, entonces el propio sistema deja que la información se lea y se escriba libremente a éste. En realidad el dispositivo puede ser un archivo, una partición del disco (espacio destinado para tal fin), o parte de la misma memoria, pero si el sistema operativo lo reconoce como un disco normal, el dispositivo logrará hacer lo que sea, siempre y cuando este en los límites de su propio espacio definido, lo que nos lleva al corazón de la compresión en tiempo real utilizando la interacción del sistema operativo y dispositivos externos.

Las peticiones de lectura y escritura al manejador de dispositivo para cada producto se manejan como cualquier otra transacción del sistema operativo. Pero dentro de cada espacio del dispositivo, se comprime o expande la información de acuerdo a la arquitectura del algoritmo.

2.3 HISTORIA DE LA COMPRESIÓN DE DATOS

La historia de la compresión no puede decirse cuando comenzó, pero podemos citar que en la época de los Romanos lo utilizaron en la representación de sus números, esto es, al momento de utilizar el símbolo "V" para indicar al número cinco fue un gran logro, puesto que no utilizaron cinco veces su unidad "IIIII" para representarlo, y éste no es sólo el único caso; otro claro ejemplo sería la representación de palabras (las que comunmente utilizamos para comunicarnos por escrito) las secretarias lo han utilizado para tomar notas y apuntes a una gran velocidad llamado TAQUIGRAFIA.

Pero entremos directamente al área de la computación, aquí basados en los antecedentes anteriores y preocupados por utilizar un espacio menor al existente, dos profesores Israelíes Abraham Lempel y Jacob Ziv desarrollaron un algoritmo capaz de reducir la longitud de los archivos y así disponer de un mayor espacio. El algoritmo de compresión que ellos publicaron en 1977 se conoce ahora como el algoritmo Lempel-Ziv y éste algoritmo es el origen de esta técnica. Varios autores hoy en día, tratan de encontrar nuevos algoritmos que tengan la característica de ser mas rápidos que sus antecesores y aún mas seguros.

Poco después, en la primavera de 1988, Haruhiko Okumura operador de sistemas y encargado de una de las principales redes del Japón en el foro de ciencias SIG de PC-VAN escribió un programa simple de compresión de datos llamado LZSS en lenguaje C, que posteriormente fue actualizado por el mismo foro de ciencias. Este programa fue basado en la versión del algoritmo de Lempel y Ziv, y fue modificado superficialmente por Storer y Szymanski. El problema que presentaba éste último algoritmo consistía en que la compresión no se llevaba a cabo cuando se deseaba comprimir demasiados archivos.

Kazuhito Miki reescribió el algoritmo LZSS en turbo Pascal y lenguaje ensamblador, con ésto desarrolló un compresor que lograba comprimir suficientes archivos en uno solo, y lo llamó LARC

En las primeras versiones de éstos algoritmos LZSS y LARC se caracterizaba por su promedio bajo en compresión. Así que se reescribió nuevamente el programa LZSS usando un método denominado árbol binario, que consiste en guardar tanto la longitud y posición de la cadena a comprimir. Aunque LARC se caracterizó por su lentitud en el momento de codificar los archivos, era sustancialmente rápido en el momento de extraerlos o decodificarlos. Así éste, algoritmo era tan sencillo que fue posible crear archivos que podían extraerse a si mismos, esto es, en el mismo archivo que contenía la información codificada se encontraba el decodificador, logrando un tamaño tan reducido que se comparaba con el tamaño ocupado de la misma informacion comprimida sin el decodificador.

Poco después, algunos programadores aficionados que sobresalían en el forum, evaluaban los compresores y sugerían las modificaciones pertinentes, por lo que LARC tenía una revisión constante. Estos programas fueron dados a conocer en los diferentes foros de PC-VAN y otras redes donde fueron bien recibidos.

En el verano de 1988 se escribió otro programa, LZARI por Haruhiko Okumura, el cual combinaba los algoritmos LZSS con una compresión aritmética que le adaptó. Sin embargo fue mas lento que LZSS.

Miki, el autor de LARC, actualizó LZARI. Aquí Haruyasu Yoshizaki remplazó el código aritmético de LZARI con una versión del codificador de Huffman que incrementó su velocidad, y lo llamo LZHUF, sin embargo él desarrolló otro algoritmo llamado LHarc que en la actualidad es el más común entre los compresores.

La seguridad en los algoritmos fue muy importante así como su sencillez. Ahora éstos productos ya resultan muy versátiles con una amplia gama de opciones.

Actualmente se siguen investigando nuevas formas u opciones que permitan que el compresor sea mucho más ágil y aún más seguro.

2.4 TIPOS DE COMPRESION DE DATOS

Existen diversos tipos de compresión, y algunos autores los clasifican como gratuitos (shareware) y comprados, buenos y malos, para gráficos y no gráficos, en fin una serie de clasificaciones que depende de las necesidades del mismo usuario.

Todos los programas de compresión de datos tiene el mismo principio pero la manera como trabaja se les clasifica, nos enfocaremos a:

- Compresión en grupos
- Compresión sobre la marcha o compresión en tiempo real
- Compresión sin pérdidas
- Compresión con pérdidas

2.4.1 COMPRESION EN GRUPOS

También conocido como compresión a nivel de archivo, fue el primero en utilizarse, por lo que es la más familiar y el método más utilizado aún en nuestros días.

Este tipo de programas trabajan cuando se desea comprimir uno o más archivos a la vez. Antes de que saliera al mercado Double Space (un programa de compresión en tiempo real), el producto mas popular fue PKZip de Phil Katz, seguido por ARCPlus y LHA programas de shareware, que pueden comprimir un texto completo de 3.3 MB de información a solo 568 KB, un porcentaje de compresión impresionante de 6 a 1. Así el archivo resultante es suficientemente pequeño que se puede copiar a un disco flexible o bajarlo desde un BBS. En forma similar estas utilerías estan disponibles para Unix y Macintosh.

En éste tipo de compresión existen utilerías que permiten la autoextracción, esto es, la información y el algoritmo de la decodificación se encuentran juntos por lo que al ejecutarlo se extraen. Y aún con esta ventaja se dificulta para aquellos usuarios casuales que necesitan familiarizarse con el uso de éstos programas, que no siempre es sencillo por lo que para ellos, la compresión de datos en tiempo real es la mejor solución.

2.4.2 COMPRESION EN TIEMPO REAL

Este tipo de compresión, (también conocidos como productos de compresión en primer plano o compresión sobre la marcha), trabajan comprimiendo y expandiendo archivos al escribir o leer respectivamente, en una forma totalmente transparente al usuario y lo que es mas importante para el sistema operativo. Estos programas utilizan hardware, software o ambos para realizar la compresión ya sea de discos enteros o de porciones de éstos. Una ventaja clara de éste tipo de compresión, es la de creer que el espacio en disco en la cual trabajamos o de cualquier otra unidad se duplica.

Como se había mencionado, éste tipo de compresión comprime el archivo al salvarlo y descomprime al leerlo, gracias a que se mantiene en memoria; y para lograrlo se apoya en un dispositivo virtual, ya sea parte de la memoria, disco o ambos, comportándose bajo las disposiciones del sistema operativo. Por lo que la compresión es tan sencilla que el usuario ni se entera de su existencia.

En 1991, DR-DOS 6.0 de Digital Research, se destaca como la primer versión del sistema operativo que contiene un algoritmo de compresión en tiempo real. Posteriormente aparecieron otros productos compresores para PCs como Double Space de Microsoft; Stacker de Stac Electronics (Carlsbad, CA); XtraDrive de Informacion Tecnológica Integrada (IIT) (Santa Clara,CA); Super Stor Pro de AddStor (Menlo Park,CA) y Double Disk Gold de Vertisoft Systems (San Francisco,CA), de los cuales Microsoft fue el último en integrarse al adquirir la tecnología de compresión con Double Space.

Actualmente, con la liberación de MS-DOS 6.0 de Microsoft, millones de usuarios quienes apenas conocían la diferencia entre un drive físico y uno lógico, fueron repentinamente envueltos con ésta tecnología que sin saberlo duplicaban el espacio de su disco duro.

2.4.3 COMPRESION SIN PERDIDAS

Los primeros algoritmos que existieron trataban de reducir el volumen de los archivos para después con un proceso contrario (descompresión), lograr obtener el archivo original.

Pues bien, la compresión sin pérdidas se refiere a aquellos programas que simplemente al descomprimir un archivo previamente reducido se obtenga la información intacta, es decir, cada uno de los bytes reducidos es restaurado a su estado original. Sin una compresión sin pérdidas no se pensaría ni siquiera utilizar uno de estos productos.

A éste tipo de compresión pertenecen a los ya anteriores productos mencionados como son: PKZip, ARCPlus y LHA para los programas compresores sobre la marcha; DR-DOS 6.0, MS-DOS 6.0, Stacker, XtraDrive, Super Stor Pro y Double Disk Gold para los programas compresores en tiempo real.

2.4.4 COMPRESION CON PERDIDAS

Las aplicaciones que utilizan técnicas de compresión con pérdidas (lossy) basadas en el Grupo Conjunto de Expertos Fotográficos (JPEG), el Grupo de Expertos de Cine (MPEG), tienen amplio uso en los gráficos sofisticados. La diferencia esta en que con la compresión con pérdidas, los algoritmos descartan una parte de la información que se estima que es superflua desde el punto de vista visual. Esta estrategia produce buenos resultados de compresión y normalmente no afecta la calidad visual de una imagen, debido esto a que las imágenes gráficas están llenas de información repetidas como son los que especifican los colores o pixeles.

Pero los tipos de compresión JPEG, MPEG, Indeo y los compresores manejados por QuickTime y Video for Windows, son solo algunos que utilizan la compresión con pérdidas, esto es, los datos en algunos casos son descartados y no es posible recuperarlos cuando se someten a éste tipo de compresión. Este tipo de compresión no son aceptados para los datos que requieran guardar su integridad, como son hojas de cálculo, bases de datos y archivos de textos. Para este tipo de archivos, solo la compresión podrá hacerse si no se pierde ni un bit de información en el momento de comprimir como al descomprimir.

Ahora, sabemos que existen muchos productos de compresión y algunos de ellos son mas eficientes que otros, lo que crea la diferencia en las razones de compresión. Y las variaciones de rendimiento son función de como interactúa el manejador de dispositivo con el procesador y la velocidad del sistema, así de como se guarda la informacion en el disco comprimido y de la eficiencia con que el hardware o software comprime y expande la información. Al final todo esto será transparente para el sistema operativo y para el usuario.

La compresión y expansión de los contenidos completos de unidades de almacenamientos (discos duros, discos flexibles y unidades de cinta) a casi la velocidad normal de acceso requiere una programación muy sofisticada, no solo en cuanto a software (programas de aplicación), sino que es posible la intervención de tarjetas controladoras (hardware).

2.5 CARACTERISTICAS QUE DEBE CUMPLIR UN COMPRESOR

Las características primordiales de los programas compresores, se basan en la reunión llevada a cabo el día 24 de mayo de 1993 en Comdex/Spring, en Atlanta, Georgia, por la compañía Stac que contemplan los cinco temas mas importantes para todas las plataformas.

Y los cinco temas son:

1) **La Seguridad**: Existe un riesgo inherente en la compresión de datos, por lo que este tema es muy importante que sea atendida, así el algoritmo o código deberá ser más seguro.

Todo programa puede tener pérdidas de informacion por un mal manejo por parte del programa, la pérdida de información valiosa que no podrá recuperarse traerá problemas para cualquier usuario que con o sin experiencia tendrá que afrontarlos arriesgándose a utilizar un programa que no ofrece seguridad.

2) **Compatibilidad**: Al utilizar éste tipo de programas no deberá entrar en conflicto con otras aplicaciones existentes, ésto es, al estar ejecutándose el programa compresor no deberá bloquearse la computadora al querer cargar por ejemplo un antivirus en modo residente u cualquier otro programa que tenga ésta característica. Así como el traspaso de información de una máquina a otra sin el problema de la compatibilidad aún si éste esta trabajando con otro sistema operativo ya sea MS-DOS 3.0, 4.0, 5.0, PS/DOS, DR-DOS, etc.

3) **Desempeño**: Al usar un compresor de datos no debe disminuir la velocidad de la computadora. Actualmente en México, la mayor arquitectura utilizada son computadoras 286 y con estos equipos aún deberán ser transparentes al usuario. Esto es, como si el programa compresor no existiera.

4) Flexibilidad de uso: Actualmente existe el ambiente DOS y el ambiente Windows; así, el usuario no tendrá que preocuparse por cargar a memoria otros dispositivos para trabajar en alguna de las dos formas.

5) Protección al disco: También otra característica que deberán contener es la autoprotección del disco y autorrecuperación de datos. Algunos ejemplos serían, verificación del disco, si se localiza algun error en disco, corregirlo. Todo disco duro contiene fragmentaciones; el programa deberá leer correctamente el espacio que queda con exactitud.

La Compañía Stac Electronic ha desarrollado el Stacker y lo ha introducido al mercado hace ya varios años, por lo que sus comentarios son dignos de ser analizados. Ahora bien, si bien es cierto que la nueva versión del Sistema Operativo que lleva incluido Double Space ha logrado una gran aceptación aquí en México, también lo es, el desconcierto logrado por la pérdida de información, aunque Microsoft comenta que su algoritmo es muy seguro, los resultados ya han sido dados a conocer. La Seguridad de la información es de vital importancia por lo que día a día se trata de mejorarlos añadiendo más utilerías y actualizando otras para evitar y lograr la confiabilidad en uno de estos programas.

3. DISEÑO DE COMPRESORES

Para lograr diseñar o construir cualquier cosa ya existente siempre es necesario conocerlo totalmente, es por esto que, en éste capítulo detallaremos el funcionamiento de un programa compresor que tiene como características de ser un compresor por grupos y sin pérdidas, así como el listado de varios programas escritos en lenguaje "C".

3.1 COMO TRABAJA UN COMPRESOR.

Para éste capítulo se describirá el funcionamiento del algoritmo compresor LZW (Lempel Ziv Welch, uno de los primeros programas compresores). Y a través de pseudocódigos se verá en forma sencilla como el compresor toma un archivo y lo comprime comprendiendo así el algoritmo.

Antes de continuar será necesario comprender algunos conceptos que estarán presentes en éste capítulo.

- Archivo de entrada.- Es aquel conjunto de caracteres que están destinados a comprimirse, es decir el archivo original.

- Archivo de salida.- Es aquel conjunto de códigos generados por el programa compresor, es decir, los conjuntos de caracteres producto de la compresión almacenados en cierto espacio.

- Tabla de cadenas.- La tabla de cadenas se encarga de almacenar el conjunto de códigos que utilizará tanto el compresor como descompresor en el momento de comparar la existencia de los mismos. Este elemento entra en acción al ejecutar el programa, ya que forma parte de éste.

- Y al módulo donde se almacenará los caracteres se le denotará "VARIABLE" donde el compresor se valdrá de él para hacer las comparaciones, siendo parte de los recursos de la computadora y al iniciar deberá estar vacio.

3.1.1 TRABAJANDO CON EL COMPRESOR

En la compresión, los conjuntos de caracteres es la entrada y el conjunto de códigos es la salida. En la descompresión, los conjuntos de códigos es la entrada y el conjunto de caracteres es la salida. La tabla de cadenas es un producto de ambos, compresion y descompresion.

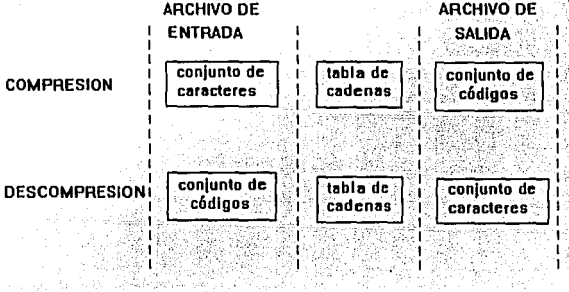


FIG 3.1 MODULOS UTILIZADOS TANTO EN LA COMPRESION COMO EN LA DESCOMPRESION

Al comprimir, el programa inicializa la tabla de cadenas, generalmente utiliza el caracter mas común en un archivo, y éste puede ser el caracter espacio (" "). La cadena puede almacenar 4096 entradas o de 0 a fff ya que utiliza un código de 12 bits. El algoritmo utiliza ésta tabla con el fin de traducir el conjunto de caracteres del archivo original, siempre tomando en cuenta una longitud fija de códigos. Esos códigos son utilizados por el algoritmo de la descompresión, (proceso contrario a la compresión), para reconstruir los datos originales.

3.1.2 COMENZANDO CON LA COMPRESION.

Al comenzar la compresión nuestra variable, anteriormente mencionada, se le adicionará el primer caracter del archivo a comprimir, ahora se buscará a través de la tabla de cadenas la existencia de éste. En caso afirmativo la variable adicionará el siguiente caracter. Ahora con esta nueva variable se busca en la tabla nuevamente; en caso de no encontrarse, se adicionará a la tabla y se comenzará con la variable vacía. Mientras que el código encontrado se graba en el archivo de salida. Así hasta finalizar el archivo.

El algoritmo se vería como esto:

[1] Se inicializa la tabla de cadenas;
[2] variable se inicializa con vacio;
[3] se adiciona a nuestra variable el próximo caracter del archivo;
[4] Se busca la variable en la tabla de cadenas. Se encuentra?

En caso afirmativo:
- Se regresa al paso [3]

En caso de no encontrarse:
- Se adiciona la variable a la tabla de cadenas;
- Se graba el código para la variable al archivo de salida;
- Se regresa al paso [2];

Así sucesivamente, hasta no existir más caracteres en el archivo a comprimir, mientras que en el archivo de salida se va grabando el código generado.

PSEUDOCODIGO

```
Procedimiento Comprimir(Archivo_entrada, Archivo_salida)
  Comienza
      Inicializa(Tabla_cadenas[ ]);
      VARIABLE <- Nulo;
       Mientras(No_fin_Arch(Archivo_entrada)
          Comienza
             VARIABLE <- VARIABLE + Leearch(Archivo_entrada);
             Si (Buscar(VARIABLE,Tabla_cadenas[ ])) entonces
```

```
            VARIABLE <- VARIABLE + Leearch(Archivo_entrada);
        Sino
            Comienza

                Inserta(VARIABLE, Tabla_cadenas[ ]);
                Código <- Caract er_encontrado;
                Escribe(Código, Archivo_salida);
                VARIABLE <- Nulo;
            Termina
        Termina
Fin.
```

El diagrama dé flujo:

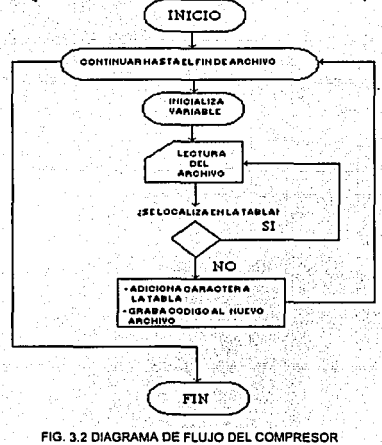


FIG. 3.2 DIAGRAMA DE FLUJO DEL COMPRESOR

3.1.3 EJEMPLO DEL COMPRESOR

Veamos un ejemplo. Supongamos tener cuatro caracteres del alfabeto: A,B,C,D. Con éstos caracteres la combinación en nuestro archivo de entrada, serían los siguientes datos: ABACABA.

Al comenzar la compresión, se inicializa la tabla de cadenas como: #0=A, #1=B, #2=C, #3=D que es el alfabeto básico. Nuestra variable tomará el primer caracter del archivo "A" y se verifica en la tabla de cadenas.

```
TABLA DE CADENAS      VARIABLE          ARCHIVO ENTRADA

   #0 --> A  <---------- 'A'  <----------  'ABACABA'
   #1 --> B
   #2 --> C
   #3 --> D
```

Como la variable si se localiza en la tabla, ahora toma el siguiente dígito y lo compara con la tabla de cadenas VARIABLE="AB" .

```
TABLA DE CADENAS
   #0 --> A
   #1 --> B     Al no encontrarse esta variable, se adiciona a la
   #2 --> C     tabla el código correspondiente a ésta variable "AB"
   #3 --> D              |
   #4 --> AB <-----------

ARCHIVO DE SALIDA     y se registra el codigo perteneciente a la variable
      #0                  "A" al archivo de salida

 VARIABLE = "B"       Ahora la variable toma sólo el último dígito y la busca en la
                        tabla de cadenas.
 VARIABLE = "BA"      Al encontrarla toma el siguiente dígito y nuevamente lo
                        compara con la tabla
```

TABLA DE CADENAS
#0 --> A
#1 --> B
#2 --> C
#3 --> D
#4 --> AB
#5 --> BA

Al no localizarse esta variable simplemente vuelve a añadirse a la tabla con el código correspondiente la variable "BA"

ARCHIVO DE SALIDA
#0,#1

Y se registra el código perteneciente a la variable ("B") al archivo de salida

VARIABLE = "A"

Ahora la variable toma sólo el último dígito y la busca en en la tabla de cadenas.

VARIABLE = "AC"

Al encontrarla toma el siguiente dígito y nuevamente lo compara con la tabla

TABLA DE CADENAS
#0 --> A
#1 --> B
#2 --> C
#3 --> D
#4 --> AB
#5 --> BA
#6 --> AC

Al no localizarse esta variable simplemente vuelve a añadirse a la tabla con el código correspondiente la variable "AC"

ARCHIVO DE SALIDA
#0,#1,#0

Y se registra el código perteneciente a la variable ("A") al archivo de salida

VARIABLE = "C"

Ahora la variable toma sólo el último dígito y la busca en la tabla de cadenas.

VARIABLE = "CA"

Al encontrarla toma el siguiente dígito y nuevamente lo compara con la tabla

TABLA DE CADENAS
#0 --> A
#1 --> B
#2 --> C
#3 --> D
#4 --> AB
#5 --> BA
#6 --> AC
#7 --> CA

Al no localizarse esta variable simplemente vuelve a añadirse a la tabla con el código correspondiente la variable "CA"

ARCHIVO DE SALIDA
#0,#1,#0,#2

Y se registra el código perteneciente a la variable ("C") al archivo de salida

VARIABLE = "A"

Ahora la variable toma sólo el último dígito y la busca en la tabla de cadenas.

VARIABLE = "AB"

Al encontrarla toma el siguiente dígito y nuevamente lo compara con la tabla

VARIABLE = "ABA"

Esta vez si encuentra la variable en la tabla y lo que hace es, tomar el siguiente dígito y volver a comparar

TABLA DE CADENAS
#0 --> A
#1 --> B
#2 --> C
#3 --> D
#4 --> AB
#5 --> BA
#6 --> AC
#7 --> CA
#8 --> ABA

Al no localizarse esta variable simplemente vuelve a añadirse a la tabla con el código correspondiente la variable "ABA"

ARCHIVO DE SALIDA
#0,#1,#0,#2,#4

Y se registra el código perteneciente a la variable ("AB") al archivo de salida

VARIABLE = "A"	Ahora este último dígito se compara con la tabla. Al encontrarla, simplemente graba el último valor al archivo de salida.
ARCHIVO DE SALIDA #0,#1,#0,#2,#4,#0	Y se registra el código perteneciente a la variable ("A") al archivo de salida

En este sencillo ejemplo, notamos que la eficiencia no fue comparativa, pero es sustancialmente eficiente si los datos de entrada tuvieran suficientes cadenas repetidas.

3.1.4 COMO TRABAJA LA DESCOMPRESION

En la descompresion, los conjuntos de códigos es la entrada y el conjunto de caracteres es la salida, como se mencionó anteriormente.

Al descomprimir, el programa inicializa la tabla de cadenas, generalmente utiliza el caracter mas común en un archivo, y éste puede ser el caracter espacio (" "). La cadena puede almacenar 4096 entradas o de 0 a fff ya que utiliza un código de 12 bits. El algoritmo utiliza ésta tabla con el fin de traducir el conjunto de códigos del archivo original.

3.1.5 COMENZANDO CON LA DESCOMPRESION.

Después, se obtiene el primer código del archivo compreso, se buscará en la tabla de cadenas en caso de encontrarlo se grabará en el archivo de salida y se obtendrá el siguiente código del archivo; si por el contrario no se localiza en la tabla de cadenas, éste se grabará al archivo de salida y se actualizará la tabla anexando éste código, para después obtener el siguiente código.

Este proceso continuará hasta el fin del archivo a comprimir;

El algoritmo queda como sigue:

1) Se inicializa la tabla de cadenas
2) Se lee el primer código del archivo de entrada
3) Se busca en la tabla de cadenas su existencia. En caso de:

   no localizado:
      a) Se actualiza la tabla de cadenas
      b) Se graba éste código en el archivo de salida
      c) Se obtiene el siguiente código del archivo de entrada

   si localizado:
      a) Se graba el caracter correspondiente al archivo de salida
      b) Se obtiene el siguiente código del archivo de entrada

4) Continuar hasta el fin del archivo de entrada

PSEUDOCODIGO

```
Procedimiento Descompresion(Archivo_entrada, Archivo_salida)
  Comienza
      Inicializa(Tabla_cadenas[ ]);
      VARIABLE <- Nulo;
      Mientras(No_fin_Arch(Archivo_entrada)
        Comienza
          VARIABLE <- VARIABLE +  Leearch(Archivo_entrada);
          Si (Buscar(VARIABLE,Tabla_cadenas[ ])) entonces
              Escribe(Variable, Archivo_salida);
              VARIABLE <- VARIABLE + Leearch(Archivo_entrada);
           Sino
              Comienza
                 Inserta(VARIABLE, Tabla_cadenas[ ]);
                 Código <- Caract er_encontrado;
                 Escribe(Código, Archivo_salida);
                 VARIABLE <- VARIABLE + Leearch(Archivo_entrada);
            Termina
        Termina
Fin.
```

El diagrama de flujo:

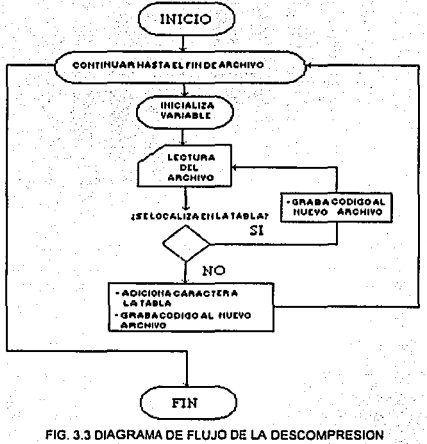


FIG. 3.3 DIAGRAMA DE FLUJO DE LA DESCOMPRESION

Viendo estos pequeños detalles en la compresion se veria muy difícil conceptualmente, pero es realmente un programa sencillo.

3.1.6 EJEMPLO DEL DESCOMPRESOR

Para el ejemplo anterior. Supongamos tener cuatro caracteres del alfabeto: A,B,C,D al iniciar la tabla. Y el archivo de entrada sean los siguientes códigos: #0,#1,#0,#2,#4,#0.

Nuestra variable tomará el primer código del archivo "#0" y se verifica en la tabla de cadenas.

TABLA DE CADENAS VARIABLE ARCHIVO ENTRADA

#0 --> A <---------- #0 <---- #0,#1,#0,#2,#4,#0
#1 --> B
#2 --> C
#3 --> D

Pero tendremos la ayuda de una segunda variable que denotaremos como "VARIABLE2" que mostrara la codificación; así el valor para ésta será "A".

VARIABLE2 = "A"

Como la VARIABLE si se localiza en la tabla de cadenas, decodifica el código y lo guarda en el archivo de salida.

ARCHIVO DE SALIDA "A" — Y se registra el caracter correspondiente al código #0 al archivo de salida

VARIABLE2 = "A" — Ahora la VARIABLE2 toma el caracter del código #0

VARIABLE = #1 — VARIABLE toma el siguiente código #1 y la busca en la tabla de cadenas

ARCHIVO DE SALIDA "AB" — Al localizarla, la decodificación se guarda en el archivo de salida, en éste caso "B"

VARIABLE2 = "AB" — Y adiciona a la VARIABLE2 el primer caracter "B"

TABLA DE CADENAS
#0 --> A
#1 --> B
#2 --> C
#3 --> D
#4 --> AB

Así mismo actualiza la tabla de cadenas con VARIABLE2 = "AB"

VARIABLE2 = "B" — Y VARIABLE2 sólo permanece con el caracter anteriormente decodificado "B"

Se lee el siguiente código siendo el #0 y nuevamente se busca en la tabla de cadenas. Como si se localiza se guarda la decodificación en el archivo de salida, así como en la VARIABLE2 y éste se adiciona a la tabla de cadenas.

Al adicionar la VARIABLE2 a la tabla de cadenas el primero se limpiará y sólo permanecerá con el último caracter decodificado, en éste caso "A".

ESTO SE VERIA ASÍ:

ARCHIVO DE SALIDA "ABA" — Al localizarse, la decodificación se guarda en el archivo de salida, en éste caso "A"

VARIABLE2 = "BA" — Y adiciona a la VARIABLE2 el caracter "A"

TABLA DE CADENAS
#0 --> A
#1 --> B
#2 --> C
#3 --> D
#4 --> AB
#5 --> BA

Así mismo actualiza la tabla de cadenas con VARIABLE2 = "BA"

VARIABLE2 = "A" — Y VARIABLE2 sólo permanece con el caracter anteriormente decodificado "A"

Se lee el siguiente código siendo el #2 y nuevamente se busca en la tabla de cadenas. Como si se localiza se guarda la decodificación en el archivo de salida, así como en la VARIABLE2 y éste se adiciona a la tabla de cadenas.

Al adicionar la VARIABLE2 a la tabla de cadenas el primero se limpiará y sólo permanecerá con el último caracter decodificado, en éste caso "C".

ESTO SE VERIA ASÍ:

ARCHIVO DE SALIDA "ABAC" — Al localizarse, la decodificación se guarda en el archivo de salida, en éste caso "C"

VARIABLE2 = "AC" — Y adiciona a la VARIABLE2 el caracter "C"

TABLA DE CADENAS
#0 --> A
#1 --> B
#2 --> C
#3 --> D
#4 --> AB
#5 --> BA
#6 --> AC

Así mismo actualiza la tabla de cadenas con VARIABLE2 = "AC"

VARIABLE2 = "C"

Y VARIABLE2 sólo permanece con el caracter anteriormente decodificado "C"

Se lee el siguiente código siendo el #4 y nuevamente se busca en la tabla de cadenas. Como si se localiza se guarda la decodificación en el archivo de salida, así como en la VARIABLE2 y éste se adiciona a la tabla de cadenas.

Al adicionar la VARIABLE2 a la tabla de cadenas el primero se limpiará y sólo permanecerá con el último caracter, en éste caso "B".

ESTO SE VERIA ASÍ:

ARCHIVO DE SALIDA
"ABACAB"

Al localizarse, la decodificación se guarda en el archivo de salida, en éste caso "AB"

VARIABLE2 = "CAB"

Y adiciona a la VARIABLE2 el caracter "AB"

TABLA DE CADENAS
#0 --> A
#1 --> B
#2 --> C
#3 --> D
#4 --> AB
#5 --> BA
#6 --> AC
#7 --> CA

Así mismo actualiza la tabla de cadenas con VARIABLE2 = "CA"

VARIABLE2 = "AB"

Y VARIABLE2 sólo permanece con el caracter anteriormente decodificado "AB"

Se lee el siguiente código siendo el #0 y nuevamente se busca en la tabla de cadenas. Como si se localiza se guarda la decodificación en el archivo de salida, así como en la VARIABLE2 y éste se adiciona a la tabla de cadenas.

Al adicionar la VARIABLE2 a la tabla de cadenas el primero se limpiará y sólo permanecerá con el último caracter, en éste caso "A".

ESTO SE VERIA ASI:

ARCHIVO DE SALIDA
"ABACABA"

Al localizarse, la decodificación se guarda en el archivo de salida, en éste caso "A"

VARIABLE2 = "ABA"

Y adiciona a la VARIABLE2 el caracter "A"

TABLA DE CADENAS
#0 --> A
#1 --> B
#2 --> C
#3 --> D
#4 --> AB
#5 --> BA
#6 --> AC
#7 --> CA
#8 --> ABA

Así mismo actualiza la tabla de cadenas con VARIABLE2 = "ABA"

VARIABLE2 = "A"

Y VARIABLE2 sólo permanece con el caracter anteriormente decodificado "A"

Existen diversos métodos que lograr el mismo objetivo; y el que acabamos de ver es sólo la forma generalizada.

Hay que notar que la forma de manejarlo corre el peligro de ocacionar un sobreflujo en la tabla de cadenas, con solo obtener un código que sobrepase el número de bits que se ha dispuesto, en nuestro ejemplo se refiere al número localizado a un lado de los códigos. En éstos casos el programador, deberá ser hábil para que ésto no ocurra.

3.2 COMPRESION JPEG

En éste tema se analizará el tipo de compresión que es utilizado en formatos gráficos siendo además un ejemplo de los compresores con pérdidas. Las características que tiene éste tipo de compresión son algo notables ya que reduce la información aún más que los compresores convencionales.

3.2.1 DEFINICION DEL COMPRESOR JPEG

Este tipo de compresión surgió de la reunión celebrada en marzo de 1991 donde varios grupos representativos tanto de hardware como de software, entre ellos C-Cube, Radius, NeXT, Storm Tech., Sun y Software Handmade; establecieron el formato JPEG como regla para el intercambio de imagenes. Y sus siglas en ingles J.P.E.G. significa Join Photografic Experts Group (Grupo de fotógrafos expertos).

JPEG que se pronuncia "yan pig" se encarga especificamente de formatos gráficos, ya que no conserva el estado original de las imagenes al momento de la decodificación. JPEG hasta ahora ha sido el regla para todo formato gráfico y trabaja básicamente con los colores y escalas de grises de las imagenes. Este tipo de compresión no sólo es utilizado para almacenar y reducir una imagen, sino también como un método para visualizar cualquier escena o película, ya sea con dibujos animados o escenas reales llamado MPEG. Y para lograr esto, es necesario utilizar una compresión con pérdidas, el cual una imagen obtenida después del proceso no es necesariamente idéntica a la imagen que fue captada en la entrada. Quizás exista cierto temor al utilizar éste método por no obtener una imagen idéntica, sin embargo, con éste método cualquier imagen, podrá reducirse de tamaño mucha más que los compresores convencionales obtendrían.

La pregunta seria ¿Porque utilizar una técnica con pérdidas si esto ocasionaría una degradación en la imagen?.

La respuesta es que éste tipo de compresión tiene el promedio mas alto que los compresores convencionales por lo que resulta conveniente utilizarla, ya que ocupa un espacio mucho menor en nuestro disco de almacenamiento, las pérdidas ocasionadas por éste método resultan aceptables. Aún cuando más se quiera cuidar estos detalles, los dispositivos de digitalización de imagenes degradan la resolución, esto es, de una imagen real o fotografía transformarla de tal manera que la computadora pueda hacer uso de ella pierde resolución; aunado a esto, los dispositivos de impresión y los monitores no ésta totalmente diseñados para obtener un resultado efectivo.

3.2.2 FUNCIONAMIENTO DEL COMPRESOR JPEG

La compresión JPEG consiste en una serie de complejas y razonables operaciones matemáticas que son los siguientes:

1. Conversión de los espacios de colores,

La imagen es convertida a un espacio de color donde separa de la imagen la intensidad y el color. Esto hecho, se debe a que la iluminación de la imagen es lo que menos importa al ojo humano que la información que llega a el a través de los colores, separando éstos factores la compresión se facilita. Esto no es una regla, ya que las imagenes en blanco y negro se hará a través de la escala de grises contenida en él.

2. Transformación discreta del coseno,

La información obtenida de la imagen (la intensidad de luz y los colores) son transformados separadamente a el dominio de la frecuencia usando una transformación discreta del coseno.

La transformación convierte las dos dimensiones espaciales ordinarias representada en un imagen dentro de otras, donde la imagen es fácilmente manejable. Los codificadores que emplean la cuantizacion por vector no usan la transformacion; en cambio, ellos simplemente reordenan los datos de la imagen para hacer la compresión más fácilmente.

En la compresión DCT (Discrete Cosine Transform) una transformacion de Fourier convierte la imagen en espacio de la frecuencia. Porque la experiencia indica que muchas imagenes naturales contienen informacion en alta frecuencia, toda la informacion de la imagen, o "energia", en el espacio de la frecuencia (el cual comprende coordenadas, dimensiones o vectores) es concrntrado dentro de componenetes de baja frecuencia (o coeficientes).

El DCT trabaja en agrupaciones de 8 x 8 pixels, el cual resulta 64 coeficientes que puede ser arreglado en un cuadrado 8 x 8 en "espacio de frecuencia"; los términos de la baja frecuencia esta muy a la izquierda, los coeficientes horizontales del alta frecuencia se incrementa de izquierda a derecha y los coeficientes verticales de altas frecuencias se incrementan de arriba hacia abajo. Para imagenes "naturales", mas de la energia (valores mas largos) miente en los pocos coeficientes de baja frecuencia en la esquina superior izquierda. Cuando no es, DCT se observa mal.

La onda de la imagen transformada, descompone la imagen dentro de imagenes espaciales multiples: por ejemplo, un filtro pasabajas o imagen promedio y un filtro paso altas o imagen diferenciada que muestra sólo los cambios rápidos o "bordos" en la imagen. Esto mantiene la informacion transformada relativamente local, habilitando mas reducción de datos selectivos a el paso de la cuantización el cual se asegura que la informacion de la alta frecuencia permanezca donde esta sea necesitada a los bordos puntiagudos.

La imagen transformada son inherentemente perdidas. Si no surge cualquier coeficiente o componente, la transformacion inversa restaura la imagen original.

3. Cuantización

Los datos transformados son cuantizados (Lo que simplemente significa que la información redundante es descartada). Esto es, si se tienen frecuencias altas se descartan y las bajas son consideradas.

El corazón de la compresión con pérdidas es la cuantizacion, en el cual pocos bits describen el mismo total cuantitativamente con la longitud "Cuantos" pasos. RGB utiliza un número de 8 bits (un coeficiente o la longitud del vector que describe 256 posibles importes de un color por pixel. La psicologia visual perspicacia de la compresión artistica quien codifica el video con poder de guiarlos para creer que en algunas imagenes ellos pueden obtener el método con solo cuatro niveles, el cual solo requiere de dos bits de informacion; así de esta manera ellos podrían descartar 6 bits, comprimiendo los datos 4 a 1.

JPEG Y MPEG estan limitados para una cuantizacion uniforme, lo cual significa que los cuatro niveles o pasos son del mismo tamaño (en este caso 64 bits en cada caso). En los codificadores tales como los vectores de cuantizacion que no permite la cuantizacion uniforme, el primero (el paso oculto) podriamos tener 128 bits originales, el segundo 64 y los últimos dos de 32 cada uno dejandolo hacer, por ejemplo la alta saturacion auzl que el azul obscuro. Al elegir los pasos de la cuantizacion que no son notables para el ojo humano es un arte.

El vector de cuantizacion trabaja en algunos pixels que al mismo tiempo en un vector tridimensional espacial en el cual cada una de los n pixels tiene tres dimensiones de color y los colores en bits representan la longitud del vector. Cuantizando estos vectores es la equivalencia de la ruptura del volumen, o n espacios dentro de simples subvolumenes.

Todas las infinitas sombras de color en la imagen original, y dicese de 128 sombras de azul obscuro en la imagen original digitalizada, son ahora reemplazados por el valor promedio en la mitad de el subvolumen. Por ejemplo en el esquema YUV9 de Indeo, 16 pixeles podrian todos empezar con la misma sombra de azul obscuro. Multiplicando 16 veces 128 posibles valores, todos serán reemplazados por uno, y se puede ver el poder del vector de cuantización, el cual basicamente reemplaza todas las imagenes posibles por un pequeño subconjunto de imagenes permitidas. Los metodos consisten en localizar la imagen permitida o de primer plano, el cual son almacenados en un look-up tablas de memoria que pueden ser cambiadas o adaptadas dependiendo de un pre-análisis de la imagen.

Usando DCT, se puede cuantizar la información en los componentes de baja frecuencia con un número relativamente largo de bits. Los cuales cuatro bits pueden describir 16 niveles uniformes, los cuales contienen 16 de los 256 posibles niveles originales. Para los coeficientes de alta frecuencia se puede cuantizar mas crudamente, utilizando solo dos bits, 1 bit o igual a nada. Si no se mantiene ningun bits de los coeficientes de frecuencia, se eliminan los componentes de frecuencia enteramente.

El arreglo de 8 x 8 de la cuantizacion DCT en el momento del tamaño es llamado una tabla Q (o tabla de calidad). Las técnicas de compresión pueden cambiar esta tabla para optimizar los resultados para diferentes tipos de imagenes. Si los codificadores eligen su propia tabla Q de una serie de tablas de busquedas o un libro de codigos basados en un pre-analisis de la imagen, esto es llamado compresión adaptiva.

Los diseñadores de Indeo, por ejemplo, sortea las imagenes dentro de cerca de 20 tipos básicos, tal como la naturaleza (cielo azul), oficinas y etc., con una tabla especial VQ para cada uno. Los técnicos trabajan en una compresión que facilite la optimizacion de la cuantizacion en escenas básicas cuadro a cuadro (o para JPEG/MPEG, siempre en bloques de 8 x 8 pixels básicos) en orden para lograr la óptima calidad. Optimage y otros compresores de alto nivel desarrollan sistemas que proporcionan herramientas tal como "Tweaking" torcedura.

4. Huffman o codificación aritmética.

Estos datos cuantizados es entonces comprimidos usando una codificador entrópico (Huffman y los códigos aritméticos)

Los codificadores utilizan tres tipos de esquemas en la compactacion. El primero es codificar la longitud, el cual reemplaza los digitos que consecutivamente son iguales con un número e identificacion (por ejemplo, 55555999999999 por 7589). El segundo es el codificador de Huffman, también llamado de longitud variable o codificador entrópico. Aquí, las cadenas de caracteres que se repiten frecuentemente son reemplazados por un código de longitud variable con la cadena mas común obteniendose así el codigo mas corto (Algo asi como código Morse). El tercero y el esquema más eficiente es el codigo aritmético, el cual compacta los datos que aparecen como un número de longitud de punto flotante que codifica la cadena común con un código de bits fraccionable.

Después de ésto, el archivo esta listo para ser almacenado como formato JPEG.

La desventaja principal de éste método, es que no se incluye la suficiente información en los archivos JPEG para que sea capaz de generar una imagen idéntica al original. Especificamente no es se incluye el espacio de color usado y la resolución de la imagen. Se esta pensando actualmente colocar la suficiente información para evitar ésta leve pérdida.

3.2.3 LA IMPORTANCIA DEL COLOR EN LAS IMAGENES

Los colores de las imagenes son normalmente almacenados en una de las siguientes formas :

1) Como un arreglo de valores de colores directos (usualmente rojo,verde y azul) (referidos a los colores de la imagen en cuestión)

2) O como un arreglo de índices dentro de un mapa de colores los cuales contienen rojo,verde y azul como valores (referidos como una paleta de colores para ese archivo).

La razón de utilizar una paleta de colores, es la de ocupar menos memoria, y así el dispositivo que desplegará la imagen será económica; con ésta paleta se cambia la cantidad de memoria usada y el poder de procesamiento en el momento de actualizarse por la velocidad menor que se obtiene. Actualmente la tarjeta TARGA 32 es un ejemplo de una tarjeta de colores diseñada especialmente para el desplegado gráfico, y una tarjeta VGA es un ejemplo de una tarjeta con Paleta.

Para cualquier desplegado gráfico deberá tomarse en cuenta los dispositivos gráficos que se tiene. Si se tiene una imagen con colores verdaderos en un dispositivo que utiliza paleta, será necesario cambiarlo a colores paleteados. Esta conversión es hecha en dos pasos: y lo primero es generar una paleta para ser usada por la imagen; y segundo tener un mapa de la imagen para la nueva paleta.

3.2.4 METODOS PARA LA GENERACION DE PALETAS

1) Un método simple y rápido es utilizar los archivos diseñados especialmente donde contienen los colores paleteados, los cuales estan uniformemente distribuidos en un cubo RGB, referido como una paleta uniforme.
Este tiene la ventaja que es mas rápido y con la misma paleta podrá ser usado por cualquier imagen; la desventaja principal es que muchas imagenes no contienen colores relacionadas con el cubo RGB, así esta paleta no podrá ser usada para la representación de colores para una imagen en particular.

El algoritmo Heckbert utilizado para la generación de la paleta de colores, un algoritmo media corto de Heckbert.

Este algoritmo primero construye una histograma tridimensional indicando como común (popular) cualquier color que lea en el cubo RGB esta en la imagen que comenzará a convertirse. Entonces procederá a subdividir este cubo histográfico (por división de cajas a la mitad) hasta que esto haya creado algunas cajas como la entrada a la paleta. La decisión como a donde dividir una caja esta basado en la distribución de colores dentro de la caja. Este algoritmo intenta crear cajas en las cuales tienen aproximadamente igual popularidad en la imagen. La entrada a la paleta estan entonces asignadas a representar caja. Existen otros métodos para la generación de una paleta de una imagen. Pero el algoritmo de Heckbert es apreciado por su rapidez y calidad.

El método generalmente utiliza el principal color principal de todos los colores en la caja (Esto le entrega el resultado matemáticamente correcto). Como quiera que sea para algunas imagenes de poca importancia mejores resultados pueden ser obtenidos por usar el centro de la caja (sin observar donde se localizan los pineal en la caja). Para imagenes con un pequeño numero de diferentes colores(menor a 16) los mejores resultados pueden ser obtenidos utilizando una esquina de la caja (las cajas cuidan la longitud cuando se reduce una imagen a un numero pequeño de colores; seleccionado un color cerca del centro de la caja podría dar una variedad de colores, pero usando la esquina de la caja te dará una saturación de colores).

3.2.5 IMAGEN MAPEADA

El próximo paso es tener de una imagen mapeada a una nueva paleta; esto es, cuando dithering se hace importante. Este simple acercamiento es un mapa de cada color en la imagen original a la paleta de entrada el cual esta cerrada a ésto.

Sin embargo, desde la entrada a la paleta generalmente representa algunos colores diferentes que la imagen original, esto resulta dentro de la banda de (Donde las áreas de cambios de colores uniformes en el original se convertirán en áreas de un solo color solido en la versión de la paleta). Esto puede ser un alivio por dithering (o modulación) los datos de la imagen tal que cualquier pineal podría no ser mapeado a esa entrada de colores cerrados, pero el promedio total de algunas áreas de la imagen puede ser cerrada a los colores correctos que podrían ser otra cosa.

3.3 COMPRENSION DE VIDEO

La compresión ha invadido generalmente todo, esto es, hacer ocupar un objeto en un espacio, lógicamente menor al que ocupa originalmente. Hasta ahora hemos estado enfocados a la compresión de archivos de datos y archivos de imagenes, pero existe un proceso más que sería una derivación de la compresión de imagenes y éste se refiere al proceso de compresión de video. La compresión del video logra comprimir una pelicula o un film (una serie de imagenes diferentes entre sí, que al verlas secuencialmente nos dará la impresión de movimiento), ocupando un espacio reducido en nuestra unidad de almacenamiento. En muchas ocaciones la capacidad de 80 MB en nuestro disco duro no es suficiente para almacenar imagenes y menos una película de unos minutos de duración. En éste capítulo se describe el proceso que se ejecuta para lograr la compresión del video (MPEG).

Para comenzar ésta explicación hay que comprender como se compone un simple cuadro de video. Un cuadro de video de buena calidad consta de 640 x 480 pixels, cada uno de los cuales tienen un color determinado de 24 bits, por lo que multiplicando 640 x 480 x 24 bits da como resultado 7,372,800 bits o 921 KB para solo un cuadro de video. Si se requiere llevar a cabo 30 imagenes por segundo, dará un total de 150 KB para el mismo tiempo (y éste es el promedio de velocidad mayor para un CD-ROM). Teniendo en cuenta ésto será necesario comprimir cada cuadro de 921 KB a menos de 5 KB para contar con el suficiente espacio para el audio y otros datos tales como texto y archivos MIDI (Sonidos). Para lograr una compresión de 200 a 1, no es tarea fácil, ya que el espectador deberá identificar las imagenes como video.

El objetivo final que pretende el compresor de video es la transmisión de la imagen a alta calidad, pero para ésto es necesario que se destinen muchos recursos, esto es, para cada cuadro requiera de al menos 1 MB de espacio. Y si aparte se desea que fuera rápido será necesario que éstos recursos se multipliquen.

Existen cuatro maneras que se pueden reducir la información de una imagen de video. Se puede reducir el número de pixels que representan la imagen (reducir el tamaño); o bien podría reducirse el número de fotos o cuadros por segundo; o bien se puede reducir el número de colores por pixel; o bien se puede comprimir los datos. Para algunos es suficiente las tres primeras opciones, pero utilizando la compresión la longitud del archivo se reducirá aún más y podrá asegurarse que la información está en su mínimo volumen.

Los algoritmos de compresión para video utilizadas para CD-ROM que no usan dispositivos especiales, ha tenido una gran demanda. Y ésto se debe a que el video es comprimido por programas tales como Super Mac de Cinepak e Intel de Indeo; ya que consiguen un promedio de 30 cuadros por segundo de 160 x 120 pixels, con colores de 16 bits y también consigue 15 cuadros por segundo con 320 x 240 pixels.

3.3.1 INICIO DE LA COMPRESION DE VIDEO

Hasta 1991 el almacenamiento del video digital, fue reproducida en el procesador intel i750, de dos maneras :

-Por video en tiempo real (RTV);
-Por video a nivel de producción (PLV)

RTV: éste codificador en tiempo real que requiere además de una tarjeta INTEL/IBM y despliega 256 x 240 pixels a 15 cuadros por segundo de una imagen digitalizada.

PLV: para éste codificador el tamaño de la imagen completa (640 x 480 pixels) es lo más importante, por lo que requiere algunos segundos para comprimir un simple cuadro, pero el video resultante es de alta calidad con 30 cuadros por segund, y el volumen de informacion se reduce considerablemente como para incorporarlo en un CD-ROM.

En ese tiempo, una colaboracion entre SuperMac Technology Inc. y Apple Computer Inc. transformaron el video analógico. Los ingenieros del software de las dos compañias desarrollaron primeramente el codificador del software que comprime el video, digitalizado en la tarjeta SuperMac de VideoSpigot la imagen, que incluia un formato de archivo sofisticado llamado QuickTime que lo desplegaba.

Los profesionales de video que siempre han trabajado con peliculas analógicas no les convence ésta nueva forma de trabajar, pero el tiempo aclarará éstas dudas. En éste momento Apple, quien recibiera los beneficios de la compresión al principio, cuenta con la mejor plataforma. Porque las peliculas en QuickTime fueron "escalables", principalmente para ellos podrian reproducirlas en cualquier Macintosh, además de tener en sincronía el video con el sonido en máquinas lentas. Los usuarios de Apple repentinamente contaban con éstos beneficios ya incluidos.

Apple sabiamente abrió la arquitectura QuickTime compartiéndolo así para aquellos que lo desearan, dando a conocer un formato de archivo que fue diseñado para operar en otras plataformas, y hacer el formato extensible. QuickTime entonces, podría buscar cada sistema de reproducción para los codificadores y seleccionar el mejor para el trabajo.

Al año, Apple presentaba QuickTime para Windows. Así Microsoft tiene un lento comienzo con la salida al mercado de Video para Windows, tomando una nueva táctica, añadiendo un acelerador de video mejorando la imagen. Diseñado para acelerar a 30 cuadros por segundo y hacer acercamientos a toda la pantalla con algunas peliculas de animación para Windows, lo que no sucede cuando el codificador es utilizado. QuickTime en la Mac permanece dominando entre los desarrolladores, pero Microsoft parece intentar alcanzarlo.

3.3.2 COMPRESION JPEG y MPEG

La compresión de video (MPEG) es una adaptación de la imagen quieta inmovil al video. En el mercado existen dos técnicas de compresión de video que fueron diseñadas por el comite MPEG que son:

a) La norma MPEG1 dicta que la reproducción al utilizar el compresor de video deberá desplegar la información de 150 KB por segundo que corresponde a la velocidad de un CD-ROM y que va de 320 x 240 pixels a 30 cuadros por segundo.

b) El segundo llamado MPEG 2 marca que la información desplegada será de 1.2 MB por segundo que corresponde a una imagen de 704 x 480 pixels a 30 cuadros por segundo. Lo que sugiere que será de mucho más calidad.

Ambos estandares requieren de un gran equipo para la reproducción.

Desafortunadamente, la compresión MPEG no esta diseñada para la edicion del video. Antes de editar el video MPEG será necesario crear un software que permita la reconstrucción en un cuadro para la edición.

Utilizacion de los tipos de compresión de video

MPEG 1	probablemente comience a ser desarrollado para manejar las imagenes y audio de CD-ROM
MPEG 2	podrá ser utilizado directamente a través de satélites de radiodifusión, televisión por cable y por productores coorporativos que necesitan una imagen de calida.

El mas reciente sistema codificador MPEG puede comprimir video en menos de 1/30 de segundo. El nuevo procesador CLM-4000 de C-Cube Microsistem puede contener en paralelo para comprimir MPEG 1 (dos chips CLM-4500) o MPEG 2 (10 chips CLM-4600) en tiempo real.

3.3.3 FUNCIONAMIENTO DE LA COMPRESION DE VIDEO

Ahora bien, la compresón de video realiza siete procesos básicos para un cuadro de video. Los primeros cuatro filtran, identifican el color, se digitaliza y se ajusta; éstos pasos son necesarios para que exista un efecto dramático en la compresión de datos; y los pasos de la compresión transforman, cuantifican y codifican la imagen.

El corazón de la compresión del video, se basa en la predicción de los movimientos, anticipándose a las imagenes. Generalmente, el compresor utiliza una imagen principal o fondo básico, en el cual solo se enfocará a comprimir aquellos pixeles que cambien con respecto a la imagen principal.

Los pasos son los siguientes:

1. FILTRACION

Para una buena compresión el primer paso es éste. La imagen original no siempre se encontrará en perfectas condiciones, por lo que el filtrado simplemente se encarga de eliminar todas aquellas irregularidades de la imagen. Este utiliza el método de difusión que consiste en obtener de la imagen los pixeles o líneas mas próximas y diferentes convirtiendola o difusionandola en otra, es decir, al tener una línea blanca junto a una negra se producirá una línea gris. Los pixeles o líneas utilizadas en el promedio se les denomina "TAPS", por ésta razón a un filtro basado en tres pixeles podrá llamarse filtro de tres taps. MPEG para un mejor promedio utiliza un filtro de siete taps.

Existen en el mercado tarjetas digitalizadoras que no utlizan éste proceso por lo cual resultan económicas pero la calidad de la imagen baja notablemente, pero aquí el compresor lo utiliza lo cual significa que es confiable para una buena calidad de la imagen.

2. CONVERSION DEL ESPACIO DE COLOR

Ya que los ojos humanos detectan menos detalles con imagenes a color en comparación con las imagenes en blanco y negro, el esquema de compresión utiliza ésta ventaja. Los codificadores lo primero que hacen es convertir los valores de los colores RBG en sus componentes de video. De ésta manera logra separar la señal blanco y negro de los colores originales.

Esto es, para la computadora un color cualquiera es descrito por tres vectores rojo, verde y azul. Esos vectores definen el color numéricamente, para lo cual un color negro sería R=G=B=0 y el blanco sería R=G=B=255.

Los componentes vectoriales definen toda la gama de colores (con la combinación del rojo, verde y azul), convirtiendo así el espacio del color.

Aún no se ha llegado al proceso final, pero con éste paso se ha reducido la longitud del archivo en promedio de 1.5 a 1

3. DIGITALIZANDO

Para digitalizar una imagen, es necesario tomar una muestra de la señal analógica por un periodo de tiempo y el resultado será un número digital. Al número de bits disponible para grabar cada valor se llama resolución (o la calidad ya sea de la imagen o sonido), y la frecuencia el cual los valores son medidos es llamado MUESTRA PROMEDIO. Un ejemplo, para un CD, se muestra un promedio de 44.1 KHz y una resolución de 16 bits; el oido humano sólo detecta sonidos a cierta frecuencia por lo que no es necesario abarcar todo o tener una resolución mas grande si el oido no lo escuchará. Para el caso del video, es necesario mostrar 9.2 millones de pixels por segundo por lo que cada muestra necesita ser de 24 bits. En éste ejemplo como en el anterior, si queremos una resolución mayor el espacio que pretendemos reducir no podrá lograrse ya que ocuparía el mismo espacio que el original.

Con la conversión del espacio en color la mitad de los colores son tipicamente descartados, lo que le conviene al proceso de digitalización.

Aún no se ha llegado al proceso final, pero con éste paso se ha reducido la longitud del archivo en un promedio de 2 a 1

4. ESCALANDO

Ahora bien, la imagen original puede ser escalable, es decir, si la imagen que proyectaremos al monitor sea menor a éste, digamos 320 x 240 pixeles, estaríamos reduciendo los datos a una cuarta parte de la imagen anterior.

Si tuviesemos un monitor que se pudiera representar 320 x 240 pixels, lograríamos reducir una cuarta parte de los datos teniendo en cuenta que existe una buena digitalización y un buen filtrado de pixeles. Cuando se muestren tan sólo 24 cuadros por segundo se podrá obtener una imagen con 256 colores que constara de 8 bits.

QuickTime en un afan de reducir aún más el espacio, hace ocupar tan sólo 160 x 120 pixeles por imagen lo que equivale a una dieciseisava parte de los datos, también reduce los cuadros por segundo a tan sólo 10 lo que equivale a una tercera parte de la información y por si fuera poco sólo muestra los colores a 8 bits que corresponde a otra tercera parte de la información original, logrando con ésto una reducción con la proporción de 16 x 3 x 3 = 144 a 1.

La desventaja de la escalabilidad es la baja calidad de la imagen en caso de no hacerse correctamente.

Aún no se ha llegado al proceso final, pero con éste paso se ha reducido la longitud del archivo en un promedio de 4 a 1

5. COMPRIMIENDO LOS DATOS

Con la información previamente preparada, llegamos a la última parte de la compresión de video; y consiste simplemente en comprimir los datos. El método de compresión más óptima, es la de pérdidas, ya que los datos que se manejan pertenecen a imagenes, por lo que el método JPEG es ideal para éste trabajo.

Al concluir éstos pasos una imagen de 1 MB de información estará concentrada en un archivo de tan sólo 24 KB. Dejando atrás a los compresores convencionales.

3.3.4 EL MERCADO DE LA COMPRESION DE VIDEO

Existen en el mercado un compresor diseñado especialmente para CD-ROM llamado MPEG 1, que logra con ésto una imagen de 320 x 240 pixels a 30 cuadros por segundo; y para lograr mayor calidad se apoya en las tarjetas aceleradoras, el cual una porcion de la imagen puede agrandarla y llenar toda la pantalla.

Recientemente, el decodificador MPEG 1 dió un salto en la evolucion de los compresores de video; para aquellos procesos que utilizaban mainframes para un proceso en tiempo real el costo era de aproximadamente de $20,000 a $75,000, ahora éste método con sólo adicionar tarjetas a las PCs se puede realizar el mismo proceso, aunque claro no habrá comparación en la calidad de la imagen.

La clave para obtener el exito de MPEG 1 depende de si los consumidores aprovechan el nuevo codificador MPEG 1 que las compañías de insumos electricos podrían introducir para el año de 1994-1995. Y con el tiempo no dudemos que puedan bajar aún más el precio de dichas tarjetas y aumentar la calidad.

Aún con el más rápido equipo Pentium y máquinas PowerPC las películas sólo pueden visualizarse en cuadros de 320 x 240 pixels y con cerca de 30 cuadros por segundo. Para llenar toda la pantalla 640 x 480 pixels se tendrá que utilizar tarjetas aceleradoras de gráficas que incluyen circuitos de acercamientos. Pero con ésto se logra dar un paso para codificar y reproducir en forma económica.

El desarrollo de MPEG no cabe duda que ha despertado curiosidad, más sin embargo, ésto es sólo el principio y las imagenes podrán ser acompañados por su propio sonido sincronizados, ayudado por CPU's más rápidos, tarjetas de aceleración de video y ¿por que no? de un mejor compresor.

3.3.5 OTROS CODIFICADORES: VECTOR DE CUANTIZACION

Cinepak, Indeo y otros más se basan en codificadores de software que utilizan una técnica de compresión rival llamada vector de cuantización. VQ es sintetizado como MPEG sin el DCT, y éste despliega la imagen por bloques y colores. A diferencia de Indeo, Cinepak no requiere un circuito especial de codificación, con una ligera desventaja que al comprimir se toma 30 veces más tiempo que la descompresión. Comprimiendo un minuto de video toma más de cinco horas en una rápida Macintoch, y dos horas de película podría llevarse un mes. Estos son sólo otras formas de solucionar el problema de la calidad de la imagen. Pero al parecer todavía falta.

PROGRAMAS DE COMPRESION

1. EL CODIFICADOR LZSS

Al primer algoritmo derivado de los estudios de Lempel y Ziv, fue modificada por Storer y Szymanski y obtuvieron lo que se le denominó codificador LZSS. Este consistía en dos partes: una de ellas era utilizar una tabla de cadenas que se inicializaba con espacios, ahí se incorporaban parte de la información en cadenas y la segunda es el utilizar un árbol binario que indicaba la longitud y posición de la información que incorporaba a la tabla.

Dado que la longitud del buffer es de 4096 bytes, la posicion puede codificarse en 12 bits. Si representamos la longitud de comparacion en cuatro bits, la longitud de posicion de la pareja es de longitud de dos bytes. Si la longitud concuerda en no mas de dos caracteres, entonces envía sólo un caracter sin codificar y comenzara el proceso con el proximo caracter. Mas, enviará un bit extra cada vez que se envía una longitud de posición o un caracter no codificado. Si la tabla se llena, se descarta el último caracter o preferiblemente el último que se usa.

La codificacion LZW ha sido adoptada por la mayoria de los programas compresores existentes tales como ARC y PKZIP.

El siguiente programa muestra éste algoritmo.

```
/*_____________________________________________________________________

Programa LZSS.C
_____________________________________________________________________*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define N               4096   /* longitud del bufer campana */
#define F                 18   /* límite superior para la longitud de comparación*/
#define THRESHOLD          2   /* cadena de comparación */
#define NIL                N   /* índice para la búsqueda en el árbol binario*/

unsigned long int
                textsize = 0,     /* longitud del texto */
                codesize = 0,     /* longitud de código */
                printcount = 0;   /* reporta el grado de avance  "cada 1K bytes" */
unsigned char
                text_buf[N + F - 1];        /* bufer campana de longitud N, con F-1 bytes extras
                                               para facilitar la comparación */

int     match_position, match_length,           /* Longitud de comparación. */
        lson[N + 1], rson[N + 257], dad[N + 1];  /* Ramas del árbol binario, izq y der. */

FILE    *infile, *outfile;              /* archivos de entrada y salida */
```

```
void InitTree(void)       /* INICIALIZA EL ARBOL */
{
        int  i;

          /* Para  i = 0 a N - 1,  rson[i] y lson[i] son los productos del nodo i
         derechos e izquierdos respectivamente. Esos nodos no serán inicializados.
         Y el arreglo dad[i] es inicializado con nulos.

          Para i = 0 a 255, rson[N + i + 1]  es la raiz del árbol para cadenas que
         comiencen con el caracter i. Estos son inicializados con nulos.  Hay que
         notar que existen 256 árboles. */

        for (i = N + 1; i <= N + 256; i++) rson[i] = NIL;
        for (i = 0; i < N; i++) dad[i] = NIL;
}

void InsertNode(int r)

          /* Inserta una cadena de longitud F, text_buf[r..r+F-1], dentro de uno de
         los árboles text_buf[r] ) y regresa la correspondiente posición y longitud a
         través de las variables globales match_position y match_length.
          Si match_length = F , entonces quita el nodo anterior .
          Hay que notar que r juega un doble papel, como nodo del árbol y posición
         en el buffer. */

{
        int  i, p, cmp;
        unsigned char  *key;

        cmp = 1;  key = &text_buf[r];  p = N + 1 + key[0];
        rson[r] = lson[r] = NIL;  match_length = 0;
        for ( ; ; ) {
                if (cmp >= 0) {
                        if (rson[p] != NIL) p = rson[p];
                        else {  rson[p] = r;  dad[r] = p;  return;  }
                } else {
                        if (lson[p] != NIL) p = lson[p];
                        else {  lson[p] = r;  dad[r] = p;  return;  }
                }
                for (i = 1; i < F; i++)
                        if ((cmp = key[i] - text_buf[p + i]) != 0)  break;
                if (i > match_length) {
                        match_position = p;
                        if ((match_length = i) >= F)  break;
                }
        }
```

```
        dad[r] = dad[p];  lson[r] = lson[p];  rson[r] = rson[p];
        dad[lson[p]] = r;  dad[rson[p]] = r;
        if (rson[dad[p]] == p) rson[dad[p]] = r;
        else                   lson[dad[p]] = r;
        dad[p] = NIL;     /* eliminando p */
}

void DeleteNode(int p)  /* borrando el nodo p del árbol */
{
        int  q;

        if (dad[p] == NIL) return;              /* no se encuentra en el árbol? */
        if (rson[p] == NIL) q = lson[p];
        else if (lson[p] == NIL) q = rson[p];
        else {
                q = lson[p];
                if (rson[q] != NIL) {
                        do {  q = rson[q];  } while (rson[q] != NIL);
                        rson[dad[q]] = lson[q];  dad[lson[q]] = dad[q];
                        lson[q] = lson[p];  dad[lson[p]] = q;
                }
                rson[q] = rson[p];  dad[rson[p]] = q;
        }

        dad[q] = dad[p];
        if (rson[dad[p]] == p) rson[dad[p]] = q;  else lson[dad[p]] = q;
        dad[p] = NIL;
}


void Encode(void)   /*  Codificación */
{
        int  i, c, len, r, s, last_match_length, code_buf_ptr;
        unsigned char  code_buf[17], mask;

        InitTree();          /* Inicializa el árbol */
        code_buf[0] = 0;

          /* la variable code_buf[1..16] guarda ocho unidades del código, y la
          variable code_buf[0]  trabaja con ocho banderas, donde "1" representa que
          la unidad es una letra no codificada (1 byte),  y "0" representa una posición y
          longitud (2 bytes). Sin embargo ocho unidades requieren de al menos 16
          bytes de codigo */

        code_buf_ptr = mask = 1;
        s = 0;  r = N - F;
        for (i = s; i < r; i++) text_buf[i] = ' ';  /* Llena el buffer con el caracter " " espacio siendo el
                                                       de mayor frecuencia */
```

```
        for (len = 0; len < F && (c = getc(infile)) != EOF; len++)
            text_buf[r + len] = c;          /* Lee F bytes dentro de los últimos bytes del buffer */
if ((textsize = len) == 0) return;      /* Verifica la existencia de texto en el archivo */


          /* Inserta F cadenas, cada uno de los cuales comienzan con uno o más el
        caracter " " (espacio).  Notese el orden en el cual esas cadenas son
        insertados.  Este método, degenera el árbol lo que resulta ser la menor de
        las ocurrencias y finalmente, inserta la cadena leída. Las variables globales
        match_length y match_position son colocados */

        for (i = 1; i <= F; i++) InsertNode(r - i);  InsertNode(r);

        do {

          /* Verifica que la variable match_length no este cerca del final del archivo. */
            if (match_length > len) match_length = len;

                if (match_length <= THRESHOLD) {
                    match_length = 1;   /* No concuerdan las longitudes, se envía un byte. */
                    code_buf[0] |= mask;  /* Envía una bandera de un byte */
                    code_buf[code_buf_ptr++] = text_buf[r];  /* Envía la decodificación */
        } else {

                      /* Envía la posición y la longitud. */

                    code_buf[code_buf_ptr++] = (unsigned char) match_position;
                    code_buf[code_buf_ptr++] = (unsigned char)
                                (((match_position >> 4) & 0xf0)
                          | (match_length - (THRESHOLD + 1)));
                }
                if ((mask <<= 1) == 0) {
                        for (i = 0; i < code_buf_ptr; i++)       /* Envía sólo 8 unidades de */
                                putc(code_buf[i], outfile);     /* codigo junto */
                        codesize += code_buf_ptr;
                        code_buf[0] = 0;  code_buf_ptr = mask = 1;
                }
                last_match_length = match_length;
                for (i = 0; i < last_match_length &&
                                (c = getc(infile)) != EOF; i++) {
                        DeleteNode(s);              /* Borra la anterior cadena y */
                        text_buf[s] = c;            /* lee los nuevos bytes */

                          /* Si la posición es cercano al fin del buffer entonces se
                          extenderá para hacer la comparación más sencilla. Desde
                          este buffer campana, se incrementa la posición  al módulo
                          N. */
```

```
                if (s < F - 1) text_buf[s + N] = c;
                s = (s + 1) & (N - 1);  r = (r + 1) & (N - 1);

                InsertNode(r);  /* Registra la cadena en text_buf[r..r+F-1] */
        }


        /* Reporta el progreso siempre que la variable texsize exceda el
        multiplo de 1024. */

        if ((textsize += i) > printcount) {
                printf("%12ld\r", textsize);  printcount += 1024;
        }

        /* Después del fin del texto no es necesario leer el archivo, pero se
        ejecuta la acción ya que el buffer podría no estar vacio */

        while (i++ < last_match_length) {
                DeleteNode(s);
                s = (s + 1) & (N - 1);  r = (r + 1) & (N - 1);
                if (--len) InsertNode(r);
        }
    } while (len > 0);/* hasta que la longitud de la cadena sea procesada */

    if (code_buf_ptr > 1) {             /* Se envía el codigo sobrante. */
        for (i = 0; i < code_buf_ptr; i++) putc(code_buf[i], outfile);
        codesize += code_buf_ptr;
    }

    /* Después de la codificación se obtiene las estadísticas del proceso. */
    printf("In : %ld bytes\n", textsize);
    printf("Out: %ld bytes\n", codesize);
    printf("Out/In: %.3f\n", (double)codesize / textsize);
}

void Decode(void)       /* LA DECODIFICACION */
{
    int  i, j, k, r, c;
    unsigned int  flags;

    for (i = 0; i < N - F; i++) text_buf[i] = ' ';
    r = N - F;  flags = 0;
    for ( ; ; ) {
        if (((flags >>= 1) & 256) == 0) {
            if ((c = getc(infile)) == EOF) break;
            flags = c | 0xff00;                 /* Utiliza un byte superior */
        }                                       /* para el contador ocho */
```

```
                if (flags & 1) {
                        if ((c = getc(infile)) == EOF) break;
                        putc(c, outfile);  text_buf[r++] = c;  r &= (N - 1);
                } else {
                        if ((i = getc(infile)) == EOF) break;
                        if ((j = getc(infile)) == EOF) break;
                        i |= ((j & 0xf0) << 4);  j = (j & 0x0f) + THRESHOLD;
                        for (k = 0; k <= j; k++) {
                                c = text_buf[(i + k) & (N - 1)];
                                putc(c, outfile);  text_buf[r++] = c;  r &= (N - 1);
                        }
                }
        }
}

int main(int argc, char *argv[])    /*  MENU PRINCIPAL */
{
        char  *s;

        if (argc != 4) {
            printf("'lzss e archivo1 archivo2' comprime el archivo1 dentro del archivo2. \n"
                   "'lzss d archivo2 archivo1' descomprime archivo2 dentro del archivo1.\n");
                return EXIT_FAILURE;
        }
        if ((s = argv[1], s[1] || strpbrk(s, "DEde") == NULL)
         || (s = argv[2], (infile  = fopen(s, "rb")) == NULL)
         || (s = argv[3], (outfile = fopen(s, "wb")) == NULL)) {
                printf("??? %s\n", s);  return EXIT_FAILURE;
        }
        if (toupper(*argv[1]) == 'E') Encode();  else Decode();
        fclose(infile);  fclose(outfile);
        return EXIT_SUCCESS;
}
```

2. CODIFICANDO CON EL CODIGO ARITMETICO

La propuesta original del concepto del codificador aritmético fue propuesto por P. Elias.

Aunque el código Huffman es óptimo que especifica que cada caracter deberá ser codificado dentro de un número fijo de bits. Un ejemplo nosotros codificaremos "AABA" usando el codigo aritmético. Para simplificar todas las suposiciones conoceremos de ante mano que las probabilidades para "A" y "B" aparecerá como 3/4 y 1/4 respectivamente

Inicialmente, consideremos el intervalo:

$$0 <= x < 1$$

Comencemos con el primer caracter es "A" el cual su probabilidad es 3/4, simplificaremos el intervalo a 3/4

$$0 <= x < 3/4$$

El próximo caracter es "A" de nuevo, así tomamos el valor 3/4

$$0 <= x < 9/16$$

El próximo es "B" el cual su probabilidad es 1/4, así tomamos el valor 1/4

$$27/64 <= x < 9/16$$

Porque "B" es el segundo elemento de nuestro alfabeto. El último caracter es "A" y el intervalo queda como:

$$27/64 <= x < 135/256$$

El cual podemos anotar una notacion binaria:

$$0.011011 <= x < 0.10000111$$

Eligiendo desde éste intervalo cualquier numero puede ser representado en pocos bits, digamos 0.1, y enviar los bits a la derecha de 0.; en este caso enviamos un bit, 1. A pesar de tener que codificar 4 letras dentro de un bit, Con el codificador de Huffman, las cuatro letras no podrian ser codificadas en menos de cuatro bits.

Al decodificar el código 1, solo revertimos el proceso: Primero, suministramos el entero "0." y a la derecha se recibe el codigo 1, resultando 0.1 en notacion binaria, o 1/2. Desde este numero es el primer 3/4 del intervalo inicial 0 <= x < 1, el primer caracter deberá ser "A". El intervalo será menor de 3/4. En este nuevo intervalo, el número 1/2 miente dentro de los 3/4 partes, así el segundo caracter es de nuevo "A" y así sucesivamente. El número de letras en el archivo original podrá hacerse separadamente (o un caracter especial de 'EOF' podrá ser anexado para indicar el fin de archivo).

El algoritmo descrito anteriormente requiere que ambos el que manda y el que recibe conozcan la probabilidad de distribucion de caracteres.

El algoritmo que se adapto borra esta restriccion por la primer suposicion uniforme o cualquier distribucion que se sobre entienda o esten de acuerdo con los caracteres que se aproximen a una distribucion verdadera, y entonces actualizando la distribución después de cada caracter que es enviado y recibido.

3. EL CODIFICADOR LZARI

En cada paso del algoritmo LZSS manda uno u otro un caracter o un par de <posicion, longitud>. Entre estos, quizas aparecera el caracter "e" con mas frecuencia que el caracter "x", y un par <posicion, longitud> de un tamaño de 3 que podría ser plebeyo que uno de longitud 18, dice. De esta manera, si codificamos el caracter mas frecuente en pocos bits y el menos frecuente en mas bits, la longitud total del texto codificado puede ser acortado. Esta consideracion sugiere si usamos el codificador Huffman o el codigo aritmetico preferiblemente o una clase de modificacion, junto con LZSS.

Esto es mas fácil decir que hacer, porque existen muchas posibles combinaciones <posicion, longitud>. El codigo adaptado puede ejecutarse estáticamente de una distribucion de frecuencias. Algunos caracteres se haran inseguros estaticos

Se amplio el caracter posicionador de 256 a tres cientos o también en su longitud y asignando a los caracteres 0 a 255 que usualmente es de 8 bits por caracter, ya que los caracteres 253 + n representa estos que sigue es una posicion de cadena de longitud n, donde n=3,4... Esta extension de los caracteres se codifican con el código aritmético modificado.

También se observa que la longitud buscada de la cadena tiende a lo principal en una lectura recientemente hecha. Para esto, la posición mas reciente podría ser codificada dentro de pocos bits. Desde la posicion 4096 son demasiados a adaptar el codificador. Se fija la probabilidad de distribución de las posiciones "manualmente". La funcion de distribución de da acompañando LZARI.C es muy tentativo; esto no tiene base experimental completa. En retrospectiva, se podría modificar el codificador a lo mas significante 6 bits o quizas por un método mucho mas ingenioso adaptando los parametros de la funcion de distribucion a las estadísticas en el momento de ejecución.

A cualquier medida, la presente version de LZARI trata la posicion quizas separadamente, así que la compresión total no significa que fuese lo óptimo. Además, la longitud de la cadena comienza arriba en el cual las cadenas son codificados dentro del par fijo <posicion, longitud> pero logicamente este valor podra cambiar de acuerdo a la longitud del par <posición, longitud> que podríamos tomar.

```
/*_____________________________________________________________

Programa LZARI.C
_____________________________________________________________*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/********** Bit I/O entrada y salida **********/

FILE *infile, *outfile;
unsigned long int  textsize = 0, codesize = 0, printcount = 0;
```

```
void Error(char *message)   /* RUTINA DE MENSAJE */
{
        printf("\n%s\n", message);
        exit(EXIT_FAILURE);
}

void PutBit(int bit)  /* Salida de un bit (bit = 0,1) */
{
        static unsigned int  buffer = 0, mask = 128;

        if (bit) buffer |= mask;
        if ((mask >>= 1) == 0) {
                if (putc(buffer, outfile) == EOF) Error("Error de Escritura");
                buffer = 0;  mask = 128;  codesize++;
        }
}

void FlushBitBuffer(void)  /* Envio de los bits sobrantes */
{
        int  i;

        for (i = 0; i < 7; i++) PutBit(0);
}

int GetBit(void)        /* Obtención de un bit ( 0 o 1 ) */
{
        static unsigned int  buffer, mask = 0;

        if ((mask >>= 1) == 0) {
                buffer = getc(infile);  mask = 128;
        }
        return ((buffer & mask) != 0);
}

/********** RUTINA LZSS con múltiples árboles binarios  **********/

#define N               4096   /* longitud del bufer campana */
#define F                 60   /* límite superior para la longitud de comparación*/
#define THRESHOLD          2   /* posición y longitud de la cadena de comparación */
#define NIL                N   /* indice para la búsqueda en el árbol binario*/

unsigned char text_buf[N + F - 1];        /* bufer campana de longitud N, con F-1 bytes extras
                                             para facilitar la comparación */

int   match_position, match_length,             /* Longitud de comparación. */
    lson[N + 1], rson[N + 257], dad[N + 1];   /* Ramas del árbol binario, izq y der. */
```

```
void InitTree(void)  /* PROCEDIMIENTO INICIALIZA ARBOLES */
{
        int  i;

        for (i = N + 1; i <= N + 256; i++) rson[i] = NIL;       /* raiz */
        for (i = 0; i < N; i++) dad[i] = NIL;  /* nodo */
}

void InsertNode(int r)  /* INSERTA EL NODO P AL ARBOL */
{
        int  i, p, cmp, temp;
        unsigned char  *key;
        cmp = 1;  key = &text_buf[r];  p = N + 1 + key[0];
        rson[r] = lson[r] = NIL;  match_length = 0;
        for ( ; ; ) {
                if (cmp >= 0) {
                        if (rson[p] != NIL) p = rson[p];
                        else {  rson[p] = r;  dad[r] = p;  return;  }
                } else {
                        if (lson[p] != NIL) p = lson[p];
                        else {  lson[p] = r;  dad[r] = p;  return;  }
                }
                for (i = 1; i < F; i++)
                        if ((cmp = key[i] - text_buf[p + i]) != 0)  break;
                if (i > THRESHOLD) {
                        if (i > match_length) {
                                match_position = (r - p) & (N - 1);
                                if ((match_length = i) >= F) break;
                        } else if (i == match_length) {
                                if ((temp = (r - p) & (N - 1)) < match_position)
                                        match_position = temp;
                        }
                }
        }
        dad[r] = dad[p];  lson[r] = lson[p];  rson[r] = rson[p];
        dad[lson[p]] = r;  dad[rson[p]] = r;
        if (rson[dad[p]] == p) rson[dad[p]] = r;
        else                   lson[dad[p]] = r;
        dad[p] = NIL;  /* borra a p */
}
```

```
void DeleteNode(int p)  /* Borra el nodo p del árbol */
{
        int  q;

        if (dad[p] == NIL) return;  /* En caso de encontrarse en el árbol */
        if (rson[p] == NIL) q = lson[p];
        else if (lson[p] == NIL) q = rson[p];
        else {
                q = lson[p];
                if (rson[q] != NIL) {
                        do {  q = rson[q];  } while (rson[q] != NIL);
                        rson[dad[q]] = lson[q];  dad[lson[q]] = dad[q];
                        lson[q] = lson[p];  dad[lson[p]] = q;
                }
                rson[q] = rson[p];  dad[rson[p]] = q;
        }
        dad[q] = dad[p];
        if (rson[dad[p]] == p) rson[dad[p]] = q;
        else                   lson[dad[p]] = q;
        dad[p] = NIL;
}

/********** COMPRESION ARITMETICA **********/

#define M   15

/* Q1 = 2 a la M que es lo suficientemente grande, pero no tan grande como  4*Q1*(Q1-1) lo que
provocaría un desbordamiento de pila */

#define Q1  (1UL << M)
#define Q2  (2 * Q1)
#define Q3  (3 * Q1)
#define Q4  (4 * Q1)
#define MAX_CUM (Q1 - 1)

#define N_CHAR  (256 - THRESHOLD + F)

unsigned long int  low = 0, high = Q4, value = 0;
int  shifts = 0;
int  char_to_sym[N_CHAR], sym_to_char[N_CHAR + 1];
unsigned int
        sym_freq[N_CHAR + 1],   /* frecuencia de simbolos */
        sym_cum[N_CHAR + 1],   /* frecuencia acumulativa de simbolos */
        position_cum[N + 1];        /* frecuencia acumulativa de posiciones */
```

```
void StartModel(void)   /* INICIALIZA MODELOS */
{
        int ch, sym, i;

        sym_cum[N_CHAR] = 0;
        for (sym = N_CHAR; sym >= 1; sym--) {
                ch = sym - 1;
                char_to_sym[ch] = sym;  sym_to_char[sym] = ch;
                sym_freq[sym] = 1;
                sym_cum[sym - 1] = sym_cum[sym] + sym_freq[sym];
        }
        sym_freq[0] = 0;    /* sentinela (diferente a sym_freq[1]) */
        position_cum[N] = 0;
        for (i = N; i >= 1; i--)
                position_cum[i - 1] = position_cum[i] + 10000 / (i + 200);
}

void UpdateModel(int sym)
{
        int i, c, ch_i, ch_sym;

        if (sym_cum[0] >= MAX_CUM) {
                c = 0;
                for (i = N_CHAR; i > 0; i--) {
                        sym_cum[i] = c;
                        c += (sym_freq[i] = (sym_freq[i] + 1) >> 1);
                }
                sym_cum[0] = c;
        }
        for (i = sym; sym_freq[i] == sym_freq[i - 1]; i--) ;
        if (i < sym) {
                ch_i = sym_to_char[i];   ch_sym = sym_to_char[sym];
                sym_to_char[i] = ch_sym;  sym_to_char[sym] = ch_i;
                char_to_sym[ch_i] = sym;  char_to_sym[ch_sym] = i;
        }
        sym_freq[i]++;
        while (--i >= 0) sym_cum[i]++;
}

static void Output(int bit)          /* Salida de 1 bit, seguido por estos componentes */
{
        PutBit(bit);
        for ( ; shifts > 0; shifts--) PutBit(! bit);
}
```

```
void EncodeChar(int ch)
{
        int  sym;
        unsigned long int  range;

        sym = char_to_sym[ch];
        range = high - low;
        high = low + (range * sym_cum[sym - 1]) / sym_cum[0];
        low +=       (range * sym_cum[sym    ]) / sym_cum[0];
        for ( ; ; ) {
                if (high <= Q2) Output(0);
                else if (low >= Q2) {
                        Output(1);  low -= Q2;  high -= Q2;
                } else if (low >= Q1 && high <= Q3) {
                        shifts++;  low -= Q1;  high -= Q1;
                } else break;
                low += low;  high += high;
        }
        UpdateModel(sym);
}

void EncodePosition(int position)
{
        unsigned long int  range;

        range = high - low;
        high = low + (range * position_cum[position    ]) / position_cum[0];
        low +=       (range * position_cum[position + 1]) / position_cum[0];
        for ( ; ; ) {
                if (high <= Q2) Output(0);
                else if (low >= Q2) {
                        Output(1);  low -= Q2;  high -= Q2;
                } else if (low >= Q1 && high <= Q3) {
                        shifts++;  low -= Q1;  high -= Q1;
                } else break;
                low += low;  high += high;
        }
}

void EncodeEnd(void)
{
        shifts++;
        if (low < Q1) Output(0);  else Output(1);
        FlushBitBuffer();
}
```

```
int BinarySearchSym(unsigned int x)
{
        int i, j, k;

        i = 1;  j = N_CHAR;
        while (i < j) {
                k = (i + j) / 2;
                if (sym_cum[k] > x) i = k + 1;  else j = k;
        }
        return i;
}

int BinarySearchPos(unsigned int x)
{
        int i, j, k;

        i = 1;  j = N;
        while (i < j) {
                k = (i + j) / 2;
                if (position_cum[k] > x) i = k + 1;  else j = k;
        }
        return i - 1;
}

void StartDecode(void)
{
        int i;

        for (i = 0; i < M + 2; i++)
                value = 2 * value + GetBit();
}

int DecodeChar(void)
{
        int         sym, ch;
        unsigned long int  range;

        range = high - low;
        sym = BinarySearchSym((unsigned int)
                (((value - low + 1) * sym_cum[0] - 1) / range));
        high = low + (range * sym_cum[sym - 1]) / sym_cum[0];
        low +=      (range * sym_cum[sym    ]) / sym_cum[0];
        for ( ; ; ) {
```

```
                if (low >= Q2) {
                        value -= Q2;  low -= Q2;  high -= Q2;
                } else if (low >= Q1 && high <= Q3) {
                        value -= Q1;  low -= Q1;  high -= Q1;
                } else if (high > Q2) break;
                low += low;  high += high;
                value = 2 * value + GetBit();
        }
        ch = sym_to_char[sym];
        UpdateModel(sym);
        return ch;
}

int DecodePosition(void)   /* DECOFICACION DE LA POSICION */
{
        int position;
        unsigned long int  range;

        range = high - low;
        position = BinarySearchPos((unsigned int)
                (((value - low + 1) * position_cum[0] - 1) / range));
        high = low + (range * position_cum[position    ]) / position_cum[0];
        low +=      (range * position_cum[position + 1]) / position_cum[0];
        for ( ; ; ) {
                if (low >= Q2) {
                        value -= Q2;  low -= Q2;  high -= Q2;
                } else if (low >= Q1 && high <= Q3) {
                        value -= Q1;  low -= Q1;  high -= Q1;
                } else if (high > Q2) break;
                low += low;  high += high;
                value = 2 * value + GetBit();
        }
        return position;
}

/********** CODIFICADOR Y DECODIFICADOR **********/

void Encode(void)  /* CODIFICADOR */
{
        int  i, c, len, r, s, last_match_length;

        fseek(infile, 0L, SEEK_END);
        textsize = ftell(infile);
        if (fwrite(&textsize, sizeof textsize, 1, outfile) < 1)
                Error("Error de escritura");  /* Mensaje de error al escribir el archivo */
        codesize += sizeof textsize;
        if (textsize == 0) return;
        rewind(infile);  textsize = 0;
```

```
        StartModel(); InitTree();
        s = 0; r = N - F;
        for (i = s; i < r; i++) text_buf[i] = ' ';
        for (len = 0; len < F && (c = getc(infile)) != EOF; len++)
                text_buf[r + len] = c;
        textsize = len;
        for (i = 1; i <= F; i++) InsertNode(r - i);
        InsertNode(r);
        do {
                if (match_length > len) match_length = len;
                if (match_length <= THRESHOLD) {
                        match_length = 1; EncodeChar(text_buf[r]);
                } else {
                        EncodeChar(255 - THRESHOLD + match_length);
                        EncodePosition(match_position - 1);
                }
                last_match_length = match_length;
                for (i = 0; i < last_match_length &&
                                (c = getc(infile)) != EOF; i++) {
                        DeleteNode(s); text_buf[s] = c;
                        if (s < F - 1) text_buf[s + N] = c;
                        s = (s + 1) & (N - 1);
                        r = (r + 1) & (N - 1);
                        InsertNode(r);
                }
                if ((textsize += i) > printcount) {
                        printf("%12ld\r", textsize); printcount += 1024;
                }
                while (i++ < last_match_length) {
                        DeleteNode(s);
                        s = (s + 1) & (N - 1);
                        r = (r + 1) & (N - 1);
                        if (--len) InsertNode(r);
                }
        } while (len > 0);
        EncodeEnd();
        printf("In : %lu bytes\n", textsize);
        printf("Out: %lu bytes\n", codesize);
        printf("Out/In: %.3f\n", (double)codesize / textsize);
}

void Decode(void)  /* DECODIFICACION */
{
        int i, j, k, r, c;
        unsigned long int  count;

        if (fread(&textsize, sizeof textsize, 1, infile) < 1)
                Error("Error al leer el archivo ");    /* Mensaje de error al leer el archivo */
```

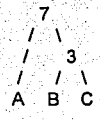
```
        if (textsize == 0) return;
        StartDecode();  StartModel();
        for (i = 0; i < N - F; i++) text_buf[i] = ' ';
        r = N - F;
        for (count = 0; count < textsize; ) {
                c = DecodeChar();
                if (c < 256) {
                        putc(c, outfile);  text_buf[r++] = c;
                        r &= (N - 1);  count++;
                } else {
                        i = (r - DecodePosition() - 1) & (N - 1);
                        j = c - 255 + THRESHOLD;
                        for (k = 0; k < j; k++) {
                                c = text_buf[(i + k) & (N - 1)];
                                putc(c, outfile);  text_buf[r++] = c;
                                r &= (N - 1);  count++;
                        }
                }
                if (count > printcount) {
                        printf("%12lu\r", count);  printcount += 1024;
                }
        }
        printf("%12lu\n", count);
}

int main(int argc, char *argv[])   /* PROCEDIMIENTO PRINCIPAL */
{
        char  *s;

        if (argc != 4) {
                printf("'lzari e archivo1 archivo2' compresion del archivo1 al archivo2.\n"
                .                "'lzari d archivo2 archivo1' descompresion del archivo2 al archivo1.\n");
                return EXIT_FAILURE;
        }
        if ((s = argv[1], s[1] || strpbrk(s, "DEde") == NULL)
         || (s = argv[2], (infile  = fopen(s, "rb")) == NULL)
         || (s = argv[3], (outfile = fopen(s, "wb")) == NULL)) {
                printf("??? %s\n", s);  return EXIT_FAILURE;
        }
        if (toupper(*argv[1]) == 'E') Encode();  else Decode();
        fclose(infile);  fclose(outfile);
        return EXIT_SUCCESS;
}
```

4. CODIFICADOR HUFFMAN

El clásico codificador Huffman es creado por Huffman. Imaginemos que el texto a codificar es "ABABACA", el cual consta de cuatro "A", dos "B" y una "C". Representamos la situación como sigue:

Combinando las dos palabras menos frecuentes en uno, el resultado es una nueva frecuencia 2 + 1 = 3 :

Se repite el paso anterior hasta obtener una combinación de caracter completamente en un arbol:



Comenzando desde el punto más alto o "raíz" de éste arbol para codificar y viajando a el caracter que se quiere codificar. Si se dirige hacia la izquierda, se manda un "0"; en caso contrario se manda un "1". Así de ésta manera, "A" es codificado por "0", "B" por 10, "C" por 11.

Así el conjunto de caracteres "ABABACA" quedaría codificado dentro de un conjunto de 10 bits "0100100110".

Al decodificar éste codigo, el decodificador deberá conocer el arbol del cual se codifico, el cual deberá ser enviado por separado.

Una modificación de éste clásico codificador de Huffman es el que se adapto, o un codigo mas dinamico. En éste método, el codificador y el decodificador procesan la primera letra de el texto como si la frecuencia de cada caracter en el archivo fuese uno, dice. Después de la primer letra que ha sido procesada, ambas partes incrementan la frecuencia de éste caracter por uno. Por ejemplo, si la primer letra es 'C', entonces la frecuencia de 'C' se hace dos, mientras que las otras frecuencias estan fijan en uno.

Por lo tanto ambas partes modifican el arbol codificador. Entonces la segunda letra podrá ser codificada y decodificada, y así sucesivamente.

5. EL CODIFICADOR LZHUF

El algoritmo LZHUF, proviene básicamente del algoritmo modificado LHarc por Haruyasu Yoshizaki, donde remplaza el codigo aritmético LZARI por el código modificado Huffman. El algoritmo LZHUF codifica a la posición mas significativa de 6 bits en los 4096 bytes del buffer buscados en la tabla. Mas recientemente, y por lo tanto mas probable las posiciones son codificadas en menos bits. En el otro lado, los restantes 6 bits son enviados iguales. Porque el codificador Huffan codifica cada letra dentro de un número fijo de bits, la búsqueda en la tabla puede facilmente implementarse.

Sin embargo teoricamente el codificador Huffman no puede exceder la compresión aritmetica, la diferencia es muy pequeña, y el codificador LZHUF es justamente rápido.

```
/*_____________________________________________________________

Programa : LZHUF.C
_____________________________________________________________*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

FILE *infile, *outfile;
unsigned long int  textsize = 0, codesize = 0, printcount = 0;
```

```
char wterr[] = "Error al escribir.";   /* Mensaje de error de escritura */

void Error(char *message)   /* PROCEDIMIENTO DE MENSAJES */
{
        printf("\n%s\n", message);
        exit(EXIT_FAILURE);
}

/********** COMPRESION LZSS **********/

#define N               4096    /* tamaño del buffer*/
#define F               60      /* tamaño del encabezado del buffer*/
#define THRESHOLD       2
#define NIL             N       /* hojas del árbol */

unsigned char
                text_buf[N + F - 1];
int             match_position, match_length,
                lson[N + 1], rson[N + 257], dad[N + 1];

void InitTree(void)     /* INICIALIZA EL ARBOL */
{
        int  i;

        for (i = N + 1; i <= N + 256; i++)
                rson[i] = NIL;                  /* raiz */
        for (i = 0; i < N; i++)
                dad[i] = NIL;                   /* nodo */
}

void InsertNode(int r)  /* PROCEDIMIENTO DE INSERCION AL ARBOL */
{
        int  i, p, cmp;
        unsigned char  *key;
        unsigned c;

        cmp = 1;
        key = &text_buf[r];
        p = N + 1 + key[0];
        rson[r] = lson[r] = NIL;
        match_length = 0;
        for ( ; ; ) {
```

```
            if (cmp >= 0) {
                    if (rson[p] != NIL)
                            p = rson[p];
                    else {
                            rson[p] = r;
                            dad[r] = p;
                            return;
                    }
            } else {
                    if (lson[p] != NIL)
                            p = lson[p];
                    else {
                            lson[p] = r;
                            dad[r] = p;
                            return;
                    }
            }
            for (i = 1; i < F; i++)
                    if ((cmp = key[i] - text_buf[p + i]) != 0)
                            break;
            if (i > THRESHOLD) {
                    if (i > match_length) {
                            match_position = ((r - p) & (N - 1)) - 1;
                            if ((match_length = i) >= F)
                                    break;
                    }
                    if (i == match_length) {
                            if ((c = ((r - p) & (N - 1)) - 1) < match_position) {
                                    match_position = c;
                            }
                    }
            }
    }
    dad[r] = dad[p];
    lson[r] = lson[p];
    rson[r] = rson[p];
    dad[lson[p]] = r;
    dad[rson[p]] = r;
    if (rson[dad[p]] == p)
            rson[dad[p]] = r;
    else
            lson[dad[p]] = r;
    dad[p] = NIL;    /* Borrando p */
}
```

```
void DeleteNode(int p)  /* ELIMINA NODO DEL ARBOL */
{
        int  q;

        if (dad[p] == NIL)
                return;                 /* en caso de no haberse registrado */
        if (rson[p] == NIL)
                q = lson[p];
        else
        if (lson[p] == NIL)
                q = rson[p];
        else {
                q = lson[p];
                if (rson[q] != NIL) {
                        do {
                                q = rson[q];
                        } while (rson[q] != NIL);
                        rson[dad[q]] = lson[q];
                        dad[lson[q]] = dad[q];
                        lson[q] = lson[p];
                        dad[lson[p]] = q;
                }
                rson[q] = rson[p];
                dad[rson[p]] = q;
        }
        dad[q] = dad[p];
        if (rson[dad[p]] == p)
                rson[dad[p]] = q;
        else
                lson[dad[p]] = q;
        dad[p] = NIL;
}

/* CODIFICACION HUFFMAN */

#define N_CHAR          (256 - THRESHOLD + F)
                                /* tipos de caracteres (donde el código = 0..N_CHAR-1) */
#define T               (N_CHAR * 2 - 1)        /* longitud de la tabla */
#define R               (T - 1)                 /* posición de la raiz */
#define MAX_FREQ        0x8000          /* actualiza el árbol cuando la frecuencia de la raiz */
                                        /* inicie con éste valor */
typedef unsigned char uchar;


/* Tabla para la codificación y decodificación superior a la posición de 6 bits */
```

```
/* Parámetros para la codificación */
uchar p_len[64] = {
        0x03, 0x04, 0x04, 0x04, 0x05, 0x05, 0x05, 0x05,
        0x05, 0x05, 0x05, 0x05, 0x06, 0x06, 0x06, 0x06,
        0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
        0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
        0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
        0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
        0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
        0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08
};

uchar p_code[64] = {
        0x00, 0x20, 0x30, 0x40, 0x50, 0x58, 0x60, 0x68,
        0x70, 0x78, 0x80, 0x88, 0x90, 0x94, 0x98, 0x9C,
        0xA0, 0xA4, 0xA8, 0xAC, 0xB0, 0xB4, 0xB8, 0xBC,
        0xC0, 0xC2, 0xC4, 0xC6, 0xC8, 0xCA, 0xCC, 0xCE,
        0xD0, 0xD2, 0xD4, 0xD6, 0xD8, 0xDA, 0xDC, 0xDE,
        0xE0, 0xE2, 0xE4, 0xE6, 0xE8, 0xEA, 0xEC, 0xEE,
        0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7,
        0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF
};

/* Y para la decodificación */
uchar d_code[256] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
        0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
        0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,
        0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
        0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
        0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
        0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
        0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
        0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
        0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
        0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A,
        0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B,
        0x0C, 0x0C, 0x0C, 0x0C, 0x0D, 0x0D, 0x0D, 0x0D,
        0x0E, 0x0E, 0x0E, 0x0E, 0x0F, 0x0F, 0x0F, 0x0F,
        0x10, 0x10, 0x10, 0x10, 0x11, 0x11, 0x11, 0x11,
        0x12, 0x12, 0x12, 0x12, 0x13, 0x13, 0x13, 0x13,
```

```
        0x14, 0x14, 0x14, 0x14, 0x15, 0x15, 0x15, 0x15,
        0x16, 0x16, 0x16, 0x16, 0x17, 0x17, 0x17, 0x17,
        0x18, 0x18, 0x19, 0x19, 0x1A, 0x1A, 0x1B, 0x1B,
        0x1C, 0x1C, 0x1D, 0x1D, 0x1E, 0x1E, 0x1F, 0x1F,
        0x20, 0x20, 0x21, 0x21, 0x22, 0x22, 0x23, 0x23,
        0x24, 0x24, 0x25, 0x25, 0x26, 0x26, 0x27, 0x27,
        0x28, 0x28, 0x29, 0x29, 0x2A, 0x2A, 0x2B, 0x2B,
        0x2C, 0x2C, 0x2D, 0x2D, 0x2E, 0x2E, 0x2F, 0x2F,
        0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
        0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F,
};

uchar d_len[256] = {
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
        0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
        0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
        0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
        0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
        0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
        0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
        0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
        0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
        0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
        0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
        0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
        0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
        0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
        0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
        0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
        0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
        0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
        0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
        0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
        0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
        0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
        0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
        0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
        0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
        0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
};

unsigned freq[T + 1];    /* Tabla de frecuencia */
```

```
int prnt[T + N_CHAR];   /* posición del nodo original, excepto para los elementos [T..T + N_CHAR
                           - 1] que son utilizados para obtener la posición de la hojas que
                           corresponde al código. */

int son[T];             /* apuntador de los nodos hijos (son[], son[] + 1) */

unsigned getbuf = 0;
uchar getlen = 0;

int GetBit(void)  /* Obtención de un bit */
{
        int i;

        while (getlen <= 8) {
                if ((i = getc(infile)) < 0) i = 0;
                getbuf |= i << (8 - getlen);
                getlen += 8;
        }
        i = getbuf;
        getbuf <<= 1;
        getlen--;
        return (i < 0);
}

int GetByte(void)           /* PROCEDIMIENTO DE OBTENCION DE UN BYTE */
{
        unsigned i;

        while (getlen <= 8) {
                if ((i = getc(infile)) < 0) i = 0;
                getbuf |= i << (8 - getlen);
                getlen += 8;
        }
        i = getbuf;
        getbuf <<= 8;
        getlen -= 8;
        return i >> 8;
}

unsigned putbuf = 0;
uchar putlen = 0;
```

```
void Putcode(int l, unsigned c)             /* Salida de c bits de codigo */
{
        putbuf |= c >> putlen;
        if ((putlen += l) >= 8) {
                if (putc(putbuf >> 8, outfile) == EOF) {
                        Error(wterr);
                }
                if ((putlen -= 8) >= 8) {
                        if (putc(putbuf, outfile) == EOF) {
                                Error(wterr);
                        }
                        codesize += 2;
                        putlen -= 8;
                        putbuf = c << (l - putlen);
                } else {
                        putbuf <<= 8;
                        codesize++;
                }
        }
}

/* INICIALIZACION DEL ARBOL */

void StartHuff(void)    /*COMIENZA CODIFICACION HUFFMAN  */
{
        int i, j;

        for (i = 0; i < N_CHAR; i++) {
                freq[i] = 1;
                son[i] = i + T;
                prnt[i + T] = i;
        }
        i = 0; j = N_CHAR;
        while (j <= R) {
                freq[j] = freq[i] + freq[i + 1];
                son[j] = i;
                prnt[i] = prnt[i + 1] = j;
                i += 2; j++;
        }
        freq[T] = 0xffff;
        prnt[R] = 0;
}
```

```
/* RECONSTRUCCION DEL ARBOL */

void reconst(void)
{
        int i, j, k;
        unsigned f, l;

        /* recoge los nodos hojas en el primer medio de la tabla */
        /* y reemplaza la frecuencia por (freq + 1) /2. */
        j = 0;
        for (i = 0; i < T; i++) {
                if (son[i] >= T) {
                        freq[j] = (freq[i] + 1) / 2;
                        son[j] = son[i];
                        j++;
                }
        }
        /* Comienza la construcción del árbol po la conexion de sus ramas */
        for (i = 0, j = N_CHAR; j < T; i += 2, j++) {
                k = i + 1;
                f = freq[j] = freq[i] + freq[k];
                for (k = j - 1; f < freq[k]; k--);
                k++;
                l = (j - k) * 2;
                memmove(&freq[k + 1], &freq[k], l);
                freq[k] = f;
                memmove(&son[k + 1], &son[k], l);
                son[k] = i;
        }

        for (i = 0; i < T; i++) {
                if ((k = son[i]) >= T) {
                        prnt[k] = i;
                } else {
                        prnt[k] = prnt[k + 1] = i;
                }
        }
}
```

```
/* Incrementa la frecuencia de obtención del código  por uno, y se actualiza el árbol */

void update(int c)
{
        int i, j, k, l;

        if (freq[R] == MAX_FREQ) {
                reconst();
        }
        c = prnt[c + T];
        do {
                k = ++freq[c];

                /* si cambia el orden de distribución cambia los nodos */
                if (k > freq[l = c + 1]) {
                        while (k > freq[++l]);
                        l--;
                        freq[c] = freq[l];
                        freq[l] = k;

                        i = son[c];
                        prnt[i] = l;
                        if (i < T) prnt[i + 1] = l;

                        j = son[l];
                        son[l] = i;

                        prnt[j] = c;
                        if (j < T) prnt[j + 1] = c;
                        son[c] = j;

                        c = l;
                }
        } while ((c = prnt[c]) != 0);           /* repetir hasta llegar a la raiz */
}

unsigned code, len;
```

```
/* Incrementa la frecuencia de obtención del código  por uno, y se actualiza el árbol */

void update(int c)
{
        int i, j, k, l;

        if (freq[R] == MAX_FREQ) {
                reconst();
        }
        c = prnt[c + T];
        do {
                k = ++freq[c];

                /* si cambia el orden de distribución cambia los nodos */
                if (k > freq[l = c + 1]) {
                        while (k > freq[++l]);
                        l--;
                        freq[c] = freq[l];
                        freq[l] = k;

                        i = son[c];
                        prnt[i] = l;
                        if (i < T) prnt[i + 1] = l;

                        j = son[l];
                        son[l] = i;

                        prnt[j] = c;
                        if (j < T) prnt[j + 1] = c;
                        son[c] = j;

                        c = l;
                }
        } while ((c = prnt[c]) != 0);           /* repetir hasta llegar a la raiz */
}

unsigned code, len;
```

```
void EncodeChar(unsigned c)     /* CODIFICACION DEL CARACTER */
{
        unsigned i;
        int j, k;

        i = 0;
        j = 0;
        k = prnt[c + T];

        /* viajando de las hojas a la raiz */
        do {
                i >>= 1;

                /* si la dirección del nodo no se reconoce, elegir el nodo inmediato superior */
                if (k & 1) i += 0x8000;

                j++;
        } while ((k = prnt[k]) != R);
        Putcode(j, i);
        code = i;
        len = j;
        update(c);
}

void EncodePosition(unsigned c)     /* CODIFICACION DE LA POSICION */
{
        unsigned i;


        /* muestra si es mayor a 6 bits localizado en la tabla */
        i = c >> 6;
        Putcode(p_len[i], (unsigned)p_code[i] << 8);

        /* muestra si es exactamente menor a 6 bits */
        Putcode(6, (c & 0x3f) << 10);
}

void EncodeEnd(void)
{
        if (putlen) {
                if (putc(putbuf >> 8, outfile) == EOF) {
                        Error(wterr);
                }
                codesize++;
        }
}
```

```
int DecodeChar(void)   /* DECODIFICACION DEL CARACTER */
{
        unsigned c;

        c = son[R];

        /* Ahora viajando de la raiz a la hojas */
        /* Elegir el nodo pequeño (en éste caso son[]) en caso de que el bit sea cero, */
        /* en caso contrario elegir el nodo mayor (son[]+1)
        while (c < T) {
                c += GetBit();
                c = son[c];
        }
        c -= T;
        update(c);
        return c;
}

int DecodePosition(void)        /* DECODIFICACION DE LA POSICION */
{
        unsigned i, j, c;

        /* recobrando los 6 bits superiores de la tabla */
        i = GetByte();
        c = (unsigned)d_code[i] << 6;
        j = d_len[i];

        /* lee exactamente 6 bits abajo */
        j -= 2;
        while (j--) {
                i = (i << 1) + GetBit();
        }
        return c | (i & 0x3f);
}

/* COMPRESION */

void Encode(void) /* PROCEDIMIENTO GENERAL DE LA CODIFICACION */
{
        int  i, c, len, r, s, last_match_length;

        fseek(infile, 0L, 2);
        textsize = ftell(infile);
        if (fwrite(&textsize, sizeof textsize, 1, outfile) < 1)
                Error(wterr);       /* En caso de que el archivo estuviera vacio */
        if (textsize == 0)
                return;
```

```
    rewind(infile);
    textsize = 0;                       /* regresar y comenzar de nuevo */
    StartHuff();
    InitTree();
    s = 0;
    r = N - F;
    for (i = s; i < r; i++)
            text_buf[i] = ' ';
    for (len = 0; len < F && (c = getc(infile)) != EOF; len++)
            text_buf[r + len] = c;
    textsize = len;
    for (i = 1; i <= F; i++)
            InsertNode(r - i);
    InsertNode(r);
    do {
            if (match_length > len)
                    match_length = len;
            if (match_length <= THRESHOLD) {
                    match_length = 1;
                    EncodeChar(text_buf[r]);
            } else {
                    EncodeChar(255 - THRESHOLD + match_length);
                    EncodePosition(match_position);
            }
            last_match_length = match_length;
            for (i = 0; i < last_match_length &&
                            (c = getc(infile)) != EOF; i++) {
                    DeleteNode(s);
                    text_buf[s] = c;
                    if (s < F - 1)
                            text_buf[s + N] = c;
                    s = (s + 1) & (N - 1);
                    r = (r + 1) & (N - 1);
                    InsertNode(r);
            }
            if ((textsize += i) > printcount) {
                    printf("%12ld\r", textsize);
                    printcount += 1024;
            }
            while (i++ < last_match_length) {
                    DeleteNode(s);
                    s = (s + 1) & (N - 1);
                    r = (r + 1) & (N - 1);
                    if (--len) InsertNode(r);
            }
    } while (len > 0);
    EncodeEnd();
```

```
        printf("In : %ld bytes\n", textsize);
        printf("Out: %ld bytes\n", codesize);
        printf("Out/In: %.3f\n", (double)codesize / textsize);
}

void Decode(void)  /* PROCEDIMIENTO DE DECODIFICACION */
{
        int  i, j, k, r, c;
        unsigned long int  count;

        if (fread(&textsize, sizeof textsize, 1, infile) < 1)
                Error("Imposible leerlo");  /* En caso de que el archivo estuviera vacio */
        if (textsize == 0)
                return;
        StartHuff();
        for (i = 0; i < N - F; i++)
                text_buf[i] = ' ';
        r = N - F;
        for (count = 0; count < textsize; ) {
                c = DecodeChar();
                if (c < 256) {
                        if (putc(c, outfile) == EOF) {
                                Error(wterr);
                        }
                        text_buf[r++] = c;
                        r &= (N - 1);
                        count++;
                } else {
                        i = (r - DecodePosition() - 1) & (N - 1);
                        j = c - 255 + THRESHOLD;
                        for (k = 0; k < j; k++) {
                                c = text_buf[(i + k) & (N - 1)];
                                if (putc(c, outfile) == EOF) {
                                        Error(wterr);
                                }
                                text_buf[r++] = c;
                                r &= (N - 1);
                                count++;
                        }
                }
                if (count > printcount) {
                        printf("%12ld\r", count);
                        printcount += 1024;
                }
        }
        printf("%12ld\n", count);
}
```

```
int main(int argc, char *argv[])    /*  RUTINA PRINCIPAL  */
{
        char  *s;

        if (argc != 4) {
                printf("'lzhuf e file1 file2' codificación del archivo1 al archivo2.\n"
                        "'lzhuf d file2 file1' decodificación del archivo2 al archivo1.\n");
                return EXIT_FAILURE;
        }
        if ((s = argv[1], s[1] || strpbrk(s, "DEde") == NULL)
         || (s = argv[2], (infile  = fopen(s, "rb")) == NULL)
         || (s = argv[3], (outfile = fopen(s, "wb")) == NULL)) {
                printf("??? %s\n", s);
                return EXIT_FAILURE;
        }
        if (toupper(*argv[1]) == 'E')
                Encode();
        else
                Decode();
        fclose(infile);
        fclose(outfile);
        return EXIT_SUCCESS;
}
```

TIPO DE COMPRESION	PARA QUE ARCHIVO?	VENTAJAS	DESVENTAJAS	PRODUCTOS COMERCIALES	DISPONIBLE
En grupo o a nivel de archivo	Para todo tipo de archivo	1. Varios archivos no comprimidos pueden estar contenidos en un solo archivo compreso. 2. Si el archivo es lo suficientemente pequeño podrá ser transportado desde un disco flexible o enviarse por medio del teléfono de una computadora a otra 3. Permiten la autoextracción	1. Al usuario casual siempre le costará trabajo aprender todas las opciones que un compresor maneja	PKZip ARCPlus LHA	DOS, UNIX, MACINTOSH
En tiempo real o sobre la marcha	Para todo tipo de archivo	1. Es transparente al usuario, trabaja al leer o escribir la información	1. Siempre éste tipo de compresor ocupará parte de la memoria convencional ocacionando que la disponibilidad de ésta no sea suficiente para otros programas. Esto se ve claramente cuando una red tiene que activarse, ya que para funcionar tiene que ocupar parte de esta memoria también, lo que ocacionaria que la disponibilidad de la memoria se agotara y resultara insuficiente para otras aplicaciones.	DR-DOS 6.0 MS-DOS 6.0 Stacker XtraDrive Super Stor Pro Double Disk Gold	DOS
Sin perdidas	Para todo tipo de archivo	1. La información que aquí se maneja es integra, lo que significa que la información que se comprime será idéntica a la información descomprimida del mismo archivo	1. En caso de utilizarlo para formatos gráficos es valido pero existen otros compresores que mejoran ésta proporción	PKZip, ARCPlus, LHA, DR-DOS 6.0, MS-DOS 6.0, Stacker, XtraDrive, Super Stor Pro y Double Disk Gold	DOS, UNIX, MACINTOSH
Con Pérdidas	Sólo para archivos con formatos gráficos	1. El porcentaje de compresión es mucho más alto que los compresores convencionales. 2. Descarta parte de la información que no se considera de importancia	1. Sólo esta disponible para formatos gráficos	JPEG MPEG Indeo QuickTime Video for Windows	DOS, MACINTOSH

CUADRO COMPARATIVO

ENTRE LOS DIFERENTES PROGRAMAS COMPRESORES MOSTRADOS EN ESTE ANEXO Y ALGUNOS TIPOS DE ARCHIVOS QUE ESTAMOS ACOSTUMBRADOS A UTILIZAR.

COMPRESORES	TIPOS DE ARCHIVOS					
	TXT	DBF	EXE	COM	PCX	BMP
LZSS	48.8	72.5	36.4	29.6	70.4	29.6
LZARI	48.8	72.5	43.7	38.2	78.0	36.4
LZHUF	54.4	79.0	43.6	38.0	78.0	36.0

COMO SE PUEDE OBSERVAR EL METODO QUE UTILIZARIAMOS SERIA EL COMPRESOR ARITMETICO POR SU ALTA COMPRESION. PERO HAY QUE NOTAR QUE DEPENDE MUCHO DEL ARCHIVO A COMPRIMIR

4. SEGURIDAD EN LA COMPRESION DE DATOS

Con la aparición de la computadora, uno de los objetivos que lo ha caracterizado es la de mantener la integridad de la información, que sin ésto no pensariamos ni siquiera en utilizarlo. En la actualidad ésta herramienta se ha extendido y se ha vuelto indispensable contar con una de ellas en cualquier área de trabajo. Hasta ahora, las fallas de éstos equipos que han reportado pérdidas de información, no ha sido por el equipo en la mayoria de las casos, sino por agentes externos a ella (fallas de luz, movimietos brúscos, etc.).

No obstante, uno de los lanzamientos de Microsoft dio la sorpresa, cuando millones de usuarios adquirieron éste producto aceptando gustosos la nueva característica, el de incorporar en el sistema operativo un algoritmo de compresión llamada Double Space, que consiste en duplicar el espacio en disco duro al comprimir y descomprimir sin que el usuario lo note.

En los primeros meses de la salida al mercado de MS-DOS 6.0 muchos usuarios se quejaban ya sea por los BBS o por los apoyos vía telefónica que Microsoft disponía, sobre los problemas que éste sistema operativo ocasionaba. Hubo inclusive quienes reportaban catastróficas pérdidas de datos, y en cambio los que no tenian problemas apoyaban con entusiasmo ésta idea.

Microsoft por su parte negaba firmemente que Double Space tuviera problemas, pero el centenar de demandas dio lugar a que en noviembre de 1993 Microsoft diera a conocer a MS-DOS 6.2 con nuevas seguridades.

Sin embargo, aún con estas seguridades, la discusión continuaba si ocacionaba o no problemas la compresión en tiempo real. Su temor se basaba en las historias de usuarios que habían perdido megabytes de valiosa información al utilizar Double Space. Si bien es cierto que la información fue irrecuperable, ésto pudo también sido ocacionado por accidentes propios del usuario. Todo estos resultados sólo indican que el software de compresión en tiempo real no presta la suficiente confianza para utilizarlo.

El problema real no radica en la compresión de datos, pero el algoritmo implementado a un sistema operativo que aún es joven y da por resultado la desconfianza. Explicando brevemente, las computadoras personales se rigen bajo ciertas normas y estandares que cubren desde las tarjetas de video hasta las líneas de entrada y salida (I/O). Pues bien, la memoria centro de los programas residentes y dispositivos propios del sistema, es la parte principal en la ejecución del programa compresor en tiempo real, que también cuenta con sus propias normas y procedimientos de acceso. Al encender la computadora son cargados aquí dichos manejadores y programas teniendo como límite 640 KB. Algunas aplicaciones utilizan el mismo BIOS para el acceso mientras que otras al momento de su instalación tiene que preparar el ambiente idoneo para su ejecución reescribiendo los archivos AUTOEXEC.BAT Y CONFIG.SYS. En otras palabras, los problemas que los usuarios han experimentado también puede ser causado por los conflictos internos del propio software de compresión con las otras partes del sistema. Sólo diré que ésto no puede ser la única causa del problema ni que el futuro de la compresión se estanque, sino por el contrario hay que trartar de mejorarlo para lograr un producto de calidad.

Una de éstas actualizaciones ha sido el de incorporar el algoritmo de compresión dentro del mismo hardware, por lo que algunas compañías han tenido de actualizar el bus de entrada y salida, así como los periféricos y aún del CPU mismo. Ya que el objetivo no es tan sólo duplicar el espacio en disco sino que además lograr el óptimo funcionamiento y dejar a tras los problemas que en un principio se enfrentaron otras compañías.

4.1. PROBLEMAS QUE DOUBLE SPACE AFRONTO

Los problemas que reportaban los usuarios en cuanto a la pérdida de información, caían en tres categorías.

1) Entre ellos, al accesar algun archivo el siguiente mensaje aparecia "Error en el archivo", que generalmente es ocacionado por algún sector defectuoso del disco duro antes de haber instalado el sistema operativo MS-DOS 6.0.

2) Otro de ellos, era la rapidez con que se llenaba el disco duro con basura y errores de disco, causados por la fragmentación del archivo al momento de comprimir.

3) En ocaciones no respondía la computadora quedandose bloqueada, ésto se debía a los conflictos internos del programa compresor y el dispositivo de MS-DOS 6.0 "Smartdrive".

Con ésta información, Microsoft respondio e incorporó nuevas seguridades al sistema operativo y lanzo al mercado el MS-DOS 6.2 junto con Double Space.

Y las utilerías incorporadas fueron:

ScanDisk : Esta utileria diagnostica y reparar automáticamente longitudes de archivos dañados y así mismo busca los errores superficiales del disco duro antes de instalar Double Space.

DoubleGuard : Una nueva utileria en proteccion, el cual avisa al usuario si otro programa residente en memoria trata de irrumpir en otra parte de la memoria RAM que ocupa Double Space.

Mientras que SmartDrive se carga en un lugar específico de la memoria, Double Space (algoritmo de compresión) pude quitarse de la memoria cuando se desee.

No obstante a pesar de los esfuerzos de Microsoft y otras compañias en incorporar nuevas seguridades, algunos usuarios han perdido la confianza en utilizar algun algorítmo de compresión en tiempo real. Y muchos, después de haber experimentado misteriosos errores al utilizar éste tipo de compresión lo han eliminado de sus computadoras, mientras que otros que ya lo tienen instalado estan temerosos por lo que pueda pasar.

Sea como fuere, más usuarios estan deseosos de tener un algorítmo capaz de duplicar el espacio disponible en sus unidades, aún con el riesgo inherente que ésto pudiera causar. Hubo inclusive usuarios que después de haber perdido toda la información no desesperaban, llegando a reformatear e instalar nuevamente el programa compresor además de sus programas de trabajo.

4.2. ORIGEN DEL PROBLEMA FUNCIONAMIENTO DE DOUBLE SPACE

Para lograr la compresión en tiempo real y mantenerla transparente como sea posible, es necesario que el proceso sea residente en memoria, esto es, que el programa siempre verifique el estado de los datos ya sea al momento de leer o escribir algún archivo. Si el software utiliza un dispositivo virtual sobre el sistema (similar a una partición lógica), implica encontrar un espacio en memoria para éste manejador del dispositivo.

A diferencia del compresor a nivel de archivo (Por ejemplo: PKZip) es que no entra en conflictos con el sistema operativo ya que éstos no se ejecutan siempre en la memoria además, el producto del proceso no es diferente a los archivos ordinarios que el mismo sistema operativo maneja. Pero los compresores en tiempo real dependen del manejador del dispositivo para asegurar la entrada/salida a través de sus rutinas de compresión.

Antes de que saliera el mercado el Sistema Opetativo MS-DOS 6.0, mas de la tercera parte de los compresores utilizaban el mismo método y era utilizar manejadores virtuales. Estos cargaban el dispositivo manejador en el archivo CONFIG.SYS y se cargaban al encender la máquina. Sin embargo, ésta acción tiene una pequeña desventaja, que es la de tener la misma letra tanto para el manejador virtual como para el manejador físico. Asi ambos manejadores aparecian como drive C, lo que necesitaban copias dupicadas del CONFIG.SYS y AUTOEXEC.BAT. Esto, nos conduce a uno de los problema de éste tipo de compresores al entrar en conflicto éstos dispositivos.

Otro problema potencial es el gasto de memoria. Si muchos dispositivos y programas residentes son cargados a la memoria convencional (los primeros 640 KB de memoria RAM), éstos pueden consumir gran parte de ésta y así entrar en conflicto con otras aplicaciones que deseen ejecutarse. Si se modifica el archivo CONFIG.SYS para cargar el manejador del compresor en la memoria superior (arriba de los 640 KB y abajo de los 1024 KB), esto podría causar conflictos con otros manejadores o programas residentes que buscan la misma localidad de memoria.

4.3. PRECARGADO DE LOS CONTROLADORES

El sistema operativo DR-DOS 6.0 de Digital Research ofreció una solución sencilla. Y consistía en nuevo archivo de sistema llamado DCONFIG.SYS que era cargado y leído antes que el archivo CONFIG.SYS. El manejador del dispositivo Super Stor puede cargarse del DCONFIG.SYS inmediatamente después del manejador de memoria, teniendo así prioridad el manejador del compresor ya que después se llama al archivo CONFIG.SYS.

Con la actualización del sistema operativo MS-DOS 6.0, Microsoft logra basicamente el mismo resultado con algunos pasos diferentes. En las versiones previas al MS-DOS no podrían cargarse un manejador antes del CONFIG.SYS, pero ahora MS-DOS 6.0 ha modificado el archivo IO.SYS del archivo de arranque que automaticamente precarga un manejador llamando DBLSPACE.BIN antes de ejecutar el CONFIG.SYS.

DBLSPACE.BIN lee un nuevo archivo de configuración llamado DBLSPACE.INI antes que cualquier otro manejador, asignando así una letra apropiada al drive y sólo entonces el control es pasado al CONFIG.SYS. El archivo CONFIG.SYS necesita ejecutar un programa llamado DBLSPACE.SYS que relocaliza a DBLSPACE.BIN de la memoria convencional. Si DBLSPACE.SYS no se ejecuta o si la entrada del CONFIG.SYS esta bloqueada, DBLSPACE.BIN permanece precargado, asegurando la permanencia en la memoria.

Una vez que el manejador del compresor fue cargado, en el sistema aparecerá como un drive virtual. Todos los archivos de entrada y salida se ejecutarán normalmente, exceptuando las intercepciones de los manejadores de entrada y salida para comprimir y descomprimir archivos, ya que ésto son guardados o cargados desde el nuevo drive virtual.

Junto a Double Space, Stacker 3.1 es sólo otro producto, otro manejador para MS-DOS que precarga éste manejador antes que CONFIG.SYS. En las versiones anteriores de Stacker había utilizado el anterior método de cargar el dispositivo dentro del CONFIG.SYS. En la actualidad sólo los productos SuperStor y Double Disk Gold del sistema Vertisoft carga sus dispositivos desde CONFIG.SYS cuando es ejecutado bajo DOS.

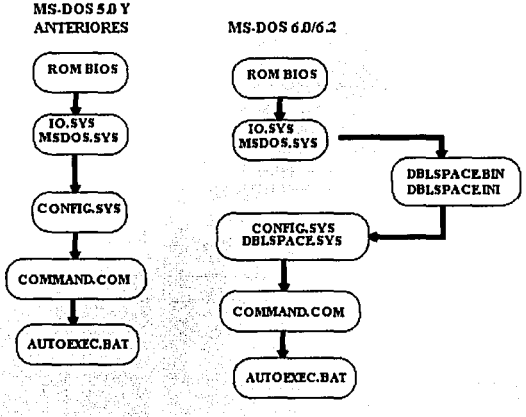


FIG. 4.1 GRAFICA A BLOQUES DEL PRECARGADO EN LA VER. 5.0 Y VER.6.2

Sin embargo, existe SuperStor/DS (Un nuevo Double Space) versión compatible de SuperStor incluido con PC-DOS 6.1 de IBM, sigue el mismo camino que Double Space, el de cargar sus manejadores antes que el archivo CONFIG:SYS. Con ésta ventaja, la última versión de DR-DOS (ahora llamado Novell DOS 7 después de la adquisición de Digital Research de Novell) ha incluido a Stacker 3.1 en lugar de SuperStor que adopta la tecnología del precargado.

4.4. ENGAÑANDO AL SISTEMA

No cabe duda que existen diferentes caminos para llegar a ROMA, el precargado fue una solución que XtraDrive no ha adoptado, ya que éste tipo de compresor carga un dispositivo manejador desde el CONFIG.SYS como SuperStor y Double Disk Gold, que manipulan los archivos de entrada y salida. Cuando se instala XtraDrive al disco duro, éste localiza el archivo de arranque del sistema operativo en otra parte sobre el manejador y susbstituye ésto con el propio archivo de arranque en el sector de arranque. Como un resultado XtraDrive arranca primero cuando se enciende la máquina y después el manejador del Sistema Operativo. Esto permite a XtraDrive interceptar las llamadas de la interrupcion 13 del BIOS (Entrada y Salida del disco) y redireccionar la Entrada y Salida (I/O) a las propias rutinas del compresor.

Claro que ésto tiene sus desventajas, al engañar al CONFIG.SYS del sistema Operativo se vuelve vulnerable. Si por alguna razón se daña el archivo CONFIG.SYS éste será ignorado y el manejador del compresor no se cargará primero. Los usuarios no tendrán pérdida de la información en éste punto, pero causará confusión al no encontrar su información ni tener acceso a ella como en la sesión anterior. Los manejadores del compresor se mantienen ahí, pero no es reconocido por el sistema operativo a menos que el problema sea corregido. En el peor de los casos, la ignorancia del usuario actue en forma destructiva y pierda sus datos actuales (como asumen que los datos fueron destruidos, reformatean el disco duro y reinstalan el software de compresión). Por esas razones, la habilidad de precargarse un manejador independiente del CONFIG.SYS es considerado una importante característica en la seguridad de los compresores en tiempo real.

4.5. SIMULACION DE UN DRIVE VIRTUAL

Otros factores de seguridad entran al juego después que los manejadores son cargados a memoria y el sistema operativo sube el manejador de compresor a la memoria alta. Todos ellos tienen algo común y es la manera de simular un drive virtual así como la organización de su estructura interna.

Por ejemplo Double Space, Stacker, SuperStor y Double Disk Gold todos simulan un drive virtual, donde almacenan la estructura interna de los archivos comprimidos; que Microsoft llama a ésto un archivo con volumen comprimido o CVF "Compressed Volumen File". Y se guarda en un único archivo en éste drive virtual. Veremos que éste archivo muestra no sólo un enorme revoltijo de datos, pero internamente el mapeo de archivos es mantenido por el software de compresión (XtraDrive de nuevo, es la excepción; éste almacena los archivos comprimidos en forma normal.)

Muchas personas temen confiar toda su información en un único archivo ya que si éste lograra dañarse toda su información se destruiría. Sin embargo, un número de seguridades y el chequeo por cruce son construidos dentro del algoritmo del compresor para prevenir la pérdida de la información y así mantenerla íntegra. Lo que nos lleva a depender de la complejidad del algoritmo de compresión (si es que cuenta con éste tipo de utilerías), para mantener la integridad de los datos.

4.6. ALMACENAMIENTO POR CLUSTERS

La manera como se almacena la información comprimida es la gran diferencia entre los programas compresores, esto es, Double Space, SuperStor, Double Disk Gold y Stacker 3.1 manipulan la información y la graban en pedazos de 8KB, éstos son comprimidos a conveniencia del tamaño de un cluster. Un cluster puede contener de 1 a 16 sectores, cada uno de ellos con 512 bytes. Pero sólo Stacker en su última versión puede subdividir un cluster y almacenar los pedazos en locaciones diversas del disco. Los otros sólo pueden almacenar un cluster en sectores que son contiguos.

Esto tiene sus desventajas y una de ellas es la perder la información si ésta es grabada en un cluster con fragmentaciones erroneas. La fragmentación inevitablemente puede suceder cada vez que alguien escriba, borre o accese la información del disco. Esto sucede con mas frecuencia bajo ciertas condiciones, pero gradualmente todo el disco comenzará a fragmentarse, especialmente si el disco esta a punto de llenarse. Consecuentemente, no existirá suficiente espacio libre que sea continuo ya sea al leer o escribir algun archivo, así DOS tiene que dividir el archivo y almacenar los clusters en otros lugares del disco. Lo que ocasionará un error si el programa compresor no localiza un cluster lo suficientemente grande para almacenar la información comprimida.

Para el caso de un disco que no cuenta con estos manejadores de compresión, no es un serio problema, ya que los clusters, aún cuando el disco esté a punto de llenarse, bastará con que existan los suficientes clusters libres para almacenar un archivo no comprimido.

En un manejador de compresión, las cosas son mas complicadas, ya que tiene que usar el espacio disponible con mayor eficiencia. Es por esto que el manejo de los clusters para almacenar la información es delicada y éstos programas deben tener mucho cuidado. Ya que en un disco no compreso cuenta con clustes de longitud fija (generalmente 8KB), cinco lineas sencillas en un archivo por lotes podría mantener ocupado a un cluster completo. En un manejador de compresión, éste archivo podría ser almacenado en un único sector con clusters de (512 bytes), así de esta manera guardandose 7.5 KB de espacio en disco. Si la información a guardar fuera de 8 KB y utilizando un compresor con promedio de 2 a 1, el archivo resultante sólo necesitaría 4 KB y ocuparía un cluster de ocho sectores (8 veces 512 bytes igual a 4 KB). En el mejor de los casos podría ser de 16 a 1 (produciendo un único sector cluster de 512 bytes). Y en el peor de los casos la compresión será de 1 a 1 sin compresión (resultando un clusters de 16 sectores, 8 KB).

Bien, pero que sucede cuando un disco esta fragmentado y DOS no puede encontrar suficientes sectores continuos para almacenar un clusters? Stacker 3.1 lo logra, reduciendo los clusters en piezas mas pequeñas (conocido como "extents" y logra guardar dicha información). Double Space, SuperStor y Double Disk Gold no pueden hacer ésto. En su lugar despliegan un error de disco lleno siempre que no exista el suficiente espacio en disco para guardar un archivo.

El problema empeora si se trata de guardar los datos que no pueden comprimirse. Tal vez un archivo previamente comprimido (compresión a nivel de archivo), o quizas un archivo con formato gráfico (jpeg). Los cluster de 8 KB no pueden ser comprimidos, por lo que es necesario 16 sectores continuos (16 veces 512 bytes igual a 8 KB), Aún si el manejador del compresor dispone de suficiente espacio en Megabytes DoubleSpace, Super Stor y Double Disk Gold no podrán guardar la información si éstos no encuentran 8 KB de sectores contiguos libres.

En teoria, el manejador de compresión podría tener cientos de megabytes libres y aún así el programa desplegará un error de disco lleno, porque no existen los suficientes clustes contiguos para almacenar la información.

Esta realidad no lleva a la conclusión de que existen todavía ciertos detalles que los creadores de los programas compresores tienen que afrontar. Es por ésto que han desarrollado en Cambrige, MA, una utileria de diagnóstico llamado DoubleCheck, el cual es un pequeño programa que muestra el problema.

En el programa de Microsoft MS-DOS 6.0 e incluso en la versión MS-DOS 6.2 han reportado que no despliega el mensaje de disco lleno y en su lugar reporta que el archivo fue guardado correctamente pero en realidad el archivo resulta inservible al accesarlo de nuevo.

Microsoft niega la existencia de éste error, pero han implementado utilerias como DEFRAG que evita todos los problemas por fragmentación, sin embargo, algunos usuarios no son muy vivaces en el mantenimiento de su sistemas, por lo que se requiere que el desfragmentador trabaje en forma automática o en tiempo real como lo hace el compresor. AddStor vende un producto llamado Double Tools para Double Space que (entre otras cosas) provee esta importante función.

4.7. UTILIZACION DE LA TABLA FAT

El compresor de Stacker a diferencia de los demás puede almacenar la información comprimida en clusters NO CONTIGUOS, teniendo un grado de eficiencia mayor. La versión de Stacker para PC (pero no en la versión para Macintosh) puede lograrlo gracias a que cuenta con un mapa adicional (FAT) que mantiene la pista de cada uno de los clusters. Esta tabla extra es una extensión de la FAT (file allocation table), en el cual el sistema operativo utiliza para la asignación de clusters.

Esta es otra área donde los productos de compresión difieren. Estos toman ligeramente un método diferente de organizar y de verificar la integridad de la FAT y relacionarlo a la estructura del mapa. Naturalmente cada vendedor declaran que su método es el más eficiente.

En forma normal, si la FAT de un disco está dañado, DOS desconocerá los clusters de datos que pertenece a los archivos, en otras palabras, la FAT es una referencia de la localización de los datos, lo que ocacionaría en caso de dañarse la pérdida total de datos, aún cuando éstos se localicen físicamente en el disco. Por lo tanto, para mayor seguridad, DOS normalmente mantiene dos copias de la FAT en un disco no comprimido. Stacker y XtraDrive también mantienen dos FAT's en el disco comprimido. Mientras que DoubleSpace, SuperStor y DoubleDisk Gold sólo mantienen una copia.

Existen argumentos que lo justifican. Y es la posibilidad de recuperar la información en caso de que la primera tabla sea dañada. Ya que la segunda obtendrá la información de la mayor parte del disco. Los daños pueden ser causados por algún desperfecto o falla de corriente cuando se ésta guardando un archivo en un disco comprimido. Una solución a ésto es que en el momento de ocurrir alguna interrupción de corriente el sistema retrase el apagado al punto de actualizar la FAT. (Uno de los cambios entre MS-DOS 6.0 y 6.2 es la utileria SmartDrive que hace esta función al apagar la máquina).

Microsoft asegura que no sólo son innecesarias las dos FAT, sino además que la tabla extra de Stacker que utiliza para subdividir la adición de clusters aumenta la complejidad a la ya compleja estructura de los programas compresores. De hecho, los productos de compresión tienen un mapa interno o tabla de referencia que es mucho mas complejo que cualquier otro programa ordinario porque contemplan tales como la longitud de los clusters variables y al mismo tiempo el porcentaje de compresión.

El aspecto preocupante es el mantenimiento que hay que darles a éstos programas, pero no hay límite para la audacia humana, ya que existen en el mercado mas de los mejores diagnósticos y utilerias de reparación que soportan a Stacker y DoubleSpace.

Todos los discos compresores traen sus propias utilerias, y esas herramientas son hechas a la medida de sus arquitecturas únicas de compresión. Frecuentemente, tratan de dar una solución sencilla a la compleja arquitectura del chequeo por cruce entre los diferentes estructuras de mapas. XtraDrive por ejemplo, compara ambas copias de los manejadores de FAT durante el encendido de la computadora. Así el programa ejecutará programa llamado VMU (Utileria para e mantenimiento del Disco "Volume Maintenance Utility) sin que el usuario lo note. El programa corrige la FAT por archivos checadores, tablas de mapeo y clusters libres.

4.8. OTROS FACTORES

En ocaciones, el programa compresor ha resultado en verdad ser un desastre en la pérdida de la información, pero hay que tomar en cuenta que éstos programas dependen en gran medida de la disposición del Sistema Operativo, y por lo tanto existen imprevistos que para lagunos serían inexplicables. Cuando existe uno de éstos problema, frecuentemente no es causado por el mismo software de compresión, más bien por interacciones entre varios elementos del sistema.

Entre ellos se encuentra:

1.- INCOMPATIBILIDAD ENTRE SOFTWARE

Microsoft ha dado a conocer una lista del software que no podría ser compatible con Double Space, que incluye copias protegidas de Lotus 123 en versiones superiores a 2.01, Informix base de datos relacional, Multimate 3.3/4.0, La versión DOS de Quicken, Movie Master 4.0, Tony La Russa Baseball, Empire Deluxe, Links, Ultima y otros. Alguno de esos programas no podrían correr en cualquier dispositivo de compresión y las razones son muy variadas, que van desde los muy difíciles esquemas de copias de protección hasta el manejo de sus archivos temporales.

2.- DEFICIENTE ADMINISTRACION EN LA ROM BIOS

Algunas versiones diferentes de la memoria ROM BIOS no trabajan apropiadamente y son causa frecuente de algunos problemas; ya que éstos no manejan apropiadamente las interrupciones que hace DBLSPACE.BIN durante el encendido de la máquina, que resulta ser un desastre para el compresor y los archivos. Ahora en la nueva versión DOS 6.2 de DBLSPACE.BIN no se efectúan interrupciones no documentadas.

3.- MEMORIA INTERMEDIA (CACHE)

A los dispositivos que disponen memoria intermedia (CACHE) para el acceso mas rápido de la información llegan a ser culpables de ésto. Algunos usuarios tienen el hábito de apagar sus computadoras inmediatamente después de haber salido de alguna aplicación y en ocaciones aún sin haber salido de ésta. Si el disco CACHE no actualizó la información antes de haber apagado la máquina los archivos abiertos podrían no haberse cerrado apropiadamente y la FAT podría no actualizarse. Este es un pequeño problema que genera con frecuencia multiples archivos corruptos. Con DOS 6.2 ahora asegura que el CACHE grabe la información antes de visualizar el prompt del sistema operativo en la pantalla, pero lo que realmente DOS necesita es un controlador de apagado, procedimiento como los que cuentan Window NT, Unix y la Mac.

4.- ERROR EN EL DISCO

Otras iteracciones como los errores del disco duro dará como consecuencia la pérdida de datos aún contando con un buen programa compresor, estos errores es posible corregirlos con el comando del sistema operativo FORMAT, algunos usuarios reparan el disco duro con utilerías que checan el disco duro los errores superficiales y son marcados como sectores dañados con el fin de no permitir el acceso a la grabación de los archivos. Los sectores dañados son marcados también en la FAT.

Este pequeño detalle lo utiliza el Sistema Operativo MS-DOS 6.2 a traves de su utilería SCANDISK antes de su instalación. La idea es limpiar y reducir cosiderablemente los errores que el disco pueda tener.

Pero que ocurre realmente cuando un archivo es almacenado en un sector dañado? pues bien, si se trata de un archivo ejecutable, el programa probablemente se interrumpa; si pertenece a una parte de un archivo de datos, la información puede perderse. Pero siempre el usuario saldrá perdiendo.

Estos son algunos de los puntos que se han sido cubiertos por Microsoft en la nueva versión del sistema operativo MS-DOS 6.2, que sin duda esta empezando abarcar éste campo por lo que simplemente se encuentra en una fase de acoplamiento.

5. EL FUTURO DE LA COMPRESION

El futuro de la compresión se define claramente, deberá ser mas transparente e infalible y para ésto será necesario que fuera mas allá del Sistema Operativo. ¿Y cuál sería ésta solución? El Hardware.

Microsoft tiene una oportunidad de oro para limpiar su nombre con la liberación de Windows 4.0 (con el nombre de CHICAGO). Como la mejor modernización del medio ambiente de la PC, Windows 4.0 podría comenzar de nuevo olvidandose de los errores cometidos en el pasado y crear una nueva fundación que tenga caracteristicas como la compresión por hardware.

Una vez más, la utilización del hardware implementado a la compresión de datos no es una idea reciente. Regresando a los dias cuando las PC's se basaban en los procesadores 8086 y 286, existió un mercado de tarjetas ISA que asombró, ya que trabajaba a través del bus de I/O, comprimiendo y descomprimiendo datos al disco duro. Los algoritmos de compresión fueron diseñados dentro de chips de alta velocidad de aquel tiempo. La compresión en tiempo real no podría basarse en software porque los CPU's no eran lo suficientemente rápidos. Hasta que surgió la rapidez del procesador 386 que podría albergar ésta posiblidad, ¡sí!, la compresión podría basarse en el software trabajando en tiempo real sin que afectara notablemente el desempeño del sistema. Por lo que las tarjetas pronto fueron olvidadas.

Más la compresión basada en Hardware ha sobrevivido en unidades de cinta, donde el proceso es seguro y transparente; y sólo algunos usuarios conocen su existencia. En efecto, QIC (Quarter-Inch Cartridge) de Stac fué quien dispuso el estandar de compresión para los respaldos en cinta a mediados de 1980.

Ahora Stac y otras compañias regresan la mirada a la compresión por hardware. Las ventajas potenciales son muchas: mejor integración con el sistema, mas transparencia al usuario, un porcentaje mayor en la compresión; mejora el desempeño de los sistemas; y quizas, rapidez en trabajo en red.

Para aumentar la velocidad del bus local han desarrollado la tecnología tal como VL-Bus y PCI (Peripheral Components Interconnect) que estan apareciendo en más PC's. Así, de esta manera solucionan las desventajas de la tarjeta ISA. La compresión por Hardware podría requerir o no intervención del usuario. Otorgando una mayor compresión gracias a la alta velocidad del chip que puede utilizar un algorítmo mas sofisticado. Lo que permite al CPU y a otros trabajos ocupar mas memoria disponible, lo que no sucede con los compresores basados en software. Finalmente; para mantener la compresión de datos en movimiento a través de la computadora y sobre redes, también pude solucionar dramáticamente el mejoramiento en forma general el desempeño del sistema.

¿Que clase de desempeño es posible que se ganaría? Stac dice tener un prototipo de tarjeta que comprime la información de 20 a 50 porciento mas rápido que los compresores basados en software y descomprime de un 10 a un 30 porciento mas rápido. Esto indica un mayor desempeño en cuanto a la compresión, y ésto sólo es el comienzo.

Los compresores por software actualmente trabajan cerca de 1 MBps en una máquina 486 de 66 Mhz, y cerca de 2 MBps en una Pentium. Para el próximo año, comenta Stac, que tendrán los chip disponibles de 10 a 20 MBps; y en 2 o tres años, de 50 a 60 MBps. Por supuesto, los CPU's también tendrán que mejorar y serán más rápidos también, pero aún queda mucho que hacer.

Los chips de compresión pueden construirse correctamente dentro una motherboard y podría acercarse a $100 dlls. en el mercado de computadoras, acordado ya con Stac. Ellos probablemente se diseñen primeramente en sistemas de alto rendimiento.

Actualmente IBM Microelectronics desarrolla a través de Ted Lattrell un chip con el algoritmo de compresión ya incluido directamente al CPU. El circuito puede llegar a comprimir datos de 40 MBps y en versiones futuras podría llegar esta cifra a 100 MBps. "Lasttrell dice que el chip tiene ya interesante atractivos para la computadora y estaciones de trabajo, quienes planean introducir estos sistemas a finales de este año o a principios de 1995".

Una vez que el primer chip sea liberado será el paso definitivo para avanzar en éste campo. Todos los desarrolladores tendrán que actualizarse ya que en caso contrario se encontrarán en desventaja en el mercado. Quizás la compresión por hardware alterará la manera de representar los datos, ésto no lo sabremos hasta que ésto suceda.

Aún existe problemas que el diseñador tendrá que afrontar como son: la adición de las aprox. 75,000 compuertas que el algoritmo llegue a utilizar. En la actualidad un procesador puede llegar a tener 625,000 compuertas de uso normal, lo que significa ampliar el espacio físico del procesador o reducir las compuertas que el algoritmo utiliza. IBM asegurá tener pronto un circuito del mismo tamaño físico y disponer de 1.3 millones de compuertas, lo que solucionaría el problema.

Desventajas

Al incluir ésta plataforma, toda la tecnología tendrá que cambiar, lo que significa que apartir de ésto, los sistemas no serán compatibles con el anterior ya que el concepto de almacenamiento será diferente.

Y por lo mismo existe otra incertidumbre, los algoritmos de compresión a nivel de archivo ¿funcionarán?, ¿tiende a desaparecer?. Esto se fundamenta en que los archivos ya compresos dentro de aquellos sistemas con el algoritmo ya incluidos en el CPU ¿podrán comprimir aún más o causarán problemas?. Pero ésto se verá después, sabemos que la disposición de los datos serán diferentes y ya habrá tiempo para pensarlo.

La única desventaja sería que éste desarrollo no será nada fácil y los desarrolladores tendrán que ingeniarselas para lograrlo.

Independientemente de ésto, las personas se maravillarán y usarán éstos algoritmos en algunos años más sin siquiera notarlo y será parte de su vida cotidiana; como sucedió con el descubrimiento del teléfono, radio, televisión, etc.; disfrutando de los beneficios sin percatarse de como surgió.

BIBLIOGRAFIA

REVISTAS

- BYTE, Vol. 19, Núm. 2 february 1994, Compression Is it safe?, Tom R. Halfhill

- COMPUTER, vol 17, Núm. 6, 1984, T.A. Welch, pp 8 - 19

- DR. DOBB'S, Núm. 219 september 1994, Color Quantization, Anton Kruger

- ELECTRONICA HOY, Núm. 6 junio 1993, Discos Duros, Autor : Greg Pastrick

- NEW MEDIA, Vol. 4 , Núm. 3 march 1994, Video Compression Crunch, Bob Doyle

- PC-COMPUTING, junio 1994, Special Issue Storage

- PC-MAGAZINE EN ESPAÑOL, Vol. 3 , Núm. 5 mayo 1992, Articulo: Triplique la capacidad de su disco duro, Autor : Greg Pastrick

- PC-MAGAZINE EN ESPAÑOL, Vol. 4 , Núm. 8 mayo 1992, Articulo: Compresión de datos, pp. 83-85

- RED, Año : III, Número : 21 , Mayo, Articulo: Los Gráficos en la Computación, Autor : Ing. Ana de la Cámara Cardero

PERIODICO

- PC-Semanal junio 93 Articulo: Compresión con Stac

LIBROS

- A tecnique for the high performance in data compression, Terry A. Welch, Ed. IEEE COMPUTER vol. 17 Num. 17, pp 8-19, June 1984

- Algorithms, R. Sedgewick, 2nd ed. Addison-Wesley, 1988

- Color Image Quantization for frame Buffer Display, Herckbert SIGGRAPH'82, proceedings, pp 297, 1982

- Communications of the ACM,I. E. Witten, R. M. Neal, and J. G. Cleary, Vol. 30, pp. 520-540 (1987),

- Data Compression, J. Ziv and A. Lempel, Ed. IEEE TRANS., pp 337-343, 1977

- Data Compression, J. Ziv and A. Lempel, Ed. IEEE TRANS., pp 530-536, 1978

- Digital Halftoning, Robert Ulichnet, Ed. MIT Press.

- Disco Duro manual de bolsillo, P.D. Moulton, Tymoty S. Stanley [Traducida al español por Roberto Escalona México,D.F.], Ed. Addison - Wesley Iberoamericana, 1990, Wilmington, Delaware, E.U.A.

- Guia para usuarios expertos, Kris Jamsa, [Traduccion : Alberto Prieto Espinosa, Ed. Osborne/McGraw-Hill, 1990

- New methods of compressing file without brute force, The Walrus, Sir++, Enricco Wizard, Ed. Transactions for the Advance of computer Sciencie vol. 45, num. 39, 787-943, March 1990

- Put a Tiger in your carl, The Walrus, Sir++, Enricco Wizard, Ed. Piratec Journal, pp 23-58, December 1990

- Virus en las Computadoras 2ª edicion, Autor : Gonzalo Ferreyra Cortez, Ed. Macrobit Editores, 1991

OTROS

- Boletin Electrónico SPIN (Sistema Profesional de Información) Tel. 628-62-00

- Image Alchemy Version 1.4, Marcos H. Woehrmann, Allan N. Hessenflow, David Kettmann, 18 de Marzo de 1991 (Documentacion del programa Alchemy de shareware)