UN COMPACTADOR USANDO EL MÉTODO DE FANO EN FORMA DINÁMICA EN UNA FUENTE DE PRIMER ORDEN

Luis G. Rueda
Raúl O. Klenzi
Nelson R. Rodriguez
Manuel O. Ortega
Franco Murciano

Docentes del Departamento de Informática - Investigadores del Instituto de Informática
Facultad de Ciencias Exactas, F. y N. - Universidad Nacional de San Juan
I.de la Roza y Meglioli (5400) San Juan, Argentina
Tel. +(54 64) 231945 - Fax +(54 64) 236940
e-mail: {lrueda,rklenzi,nelson,mortega}@iinfo.unsj.edu.ar

RESUMEN

En este trabajo se implementa el método de codificación de Fano en forma dinámica, que permite codificar y simultáneamente adaptar la codificación a los símbolos que van apareciendo. Por otro lado, se ha tomado este método como fuente de información de orden 1, es decir codificando cada símbolo actual pero teniendo en cuenta el anterior a este. A partir de estos conceptos se desarrolló un algoritmo que aprovecha las ventajas de cada enfoque. Se han realizado pruebas del algoritmo sobre distintos tipos de archivos y los resultados han sido comparados con una fuente de orden 0. Resultados muy interesantes han sido obtenidos, principalmente en archivos que presentan bajo contenido de información para una fuente de orden 1 en comparación con la de orden 0.

1. Introducción

Los métodos de codificación de Shannon, Fano [HAM80], Huffman [LIP88] y otros, en su origen implementados en forma estática, presentan varios inconvenientes por ejemplo almacenar el árbol de decodificación como cabecera de un archivo comprimido, realizar una lectura anterior a la codificación, o bien tener probabilidades históricas de las ocurrencias de los símbolos lo que no daba una buena relación de compresión; otra desventaja es que en el caso de la compresión en línea no es posible relevar la información, sino que se debe adaptar la codificación a los símbolos que van apareciendo y simultáneamente realizar la codificación.

Uno de los pasos más importantes de la Teoría de la Codificación fue la implementación de los algoritmos estáticos de codificación en forma dinámica, tales como Huffman [RUE95] o Fano [STO88]. Son varias las razones que llevaron a esto, entre las más importantes pueden mencionarse que ya no es necesario almacenar el árbol de decodificación para un lote en particular; no se necesita conocer la distribución probabilística de las ocurrencias de los símbolos a codificar, lo que permite una codificación en línea, sin necesidad de conocer que símbolos se codificaran más adelante.

Otro de los avances surgió de tomar un conjunto particular de símbolos a codificar, donde se puede observar que además de haber símbolos que ocurren con mayor frecuencia, existen símbolos que pueden ocurrir mas o menos según los anteriores, que se conoce como proceso de Markov [HAM80]. Ahora el problema es mayor, por cuanto se necesita un almacenamiento mucho mayor, por ejemplo: para un proceso de Markov de orden 1, teniendo en cuenta el conjunto de caracteres ASCII, se necesitan 256 arboles para representar la codificación de cada símbolo dependiendo del anterior. En un proceso de Markov de orden m, para cada combinación de m símbolos anteriores se tiene una codificación distinta, más precisamente, 256^{m-1} arboles de decodificación distintos.

Tomando esta última como una fuente de información de orden 1, es decir codificando el símbolo actual pero teniendo en cuenta el anterior a éste, se ha desarrollado un algoritmo que aprovecha las ventajas de cada enfoque. Se han realizado pruebas del algoritmo sobre distintos tipos de archivos en un ambiente DOS los cuales han sido comparados con una fuente de orden 0, y en el caso de los archivos que presentan un bajo contenido de información (visto como fuente de Markov de orden 1) se han obtenido muy buenos resultados, lo que posibilitaría en un futuro tomar como base fuentes de información de ordenes superiores lo que implica la obtención de resultados superiores.

2. Fuente de Información

2.1. Fuente de memoria nula

Sea $F=\{s_1,s_2,...,s_n\}$ una fuente de símbolos de un alfabeto determinado, por ejemplo, las 27 letras del alfabeto castellano; para una sucesión de símbolos:

Definición 1: Se define como peso p del símbolo s_i, y se simboliza $p(s_i)$, a la cantidad de veces que s_i se repite en la sucesión.

Ejemplo 2.1:
Sea $F = \{a,b,r,c,d\}$ una fuente, y *abracadabra* una sucesión cualquiera de la fuente F; el peso de a es $p(a) = 5$, el de b es $p(b) = 2$.

El peso de cada símbolo es un factor importante para la implementación de un código de longitud variable [ABR63]. En estos códigos es fundamental el concepto de Longitud Promedio.

Definición 2: Sea $F=\{s_1,s_2,...,s_n\}$ una fuente de símbolos de un alfabeto, $p(s_i)$ el peso del símbolo s_i y l_i la longitud del código del símbolo s_i; se define la Longitud Promedio del código y se simboliza L, así:

$$L = \frac{1}{\Omega} \sum_i p_i l_i \qquad 2.1$$

donde Ω representa el espacio muestral, es decir la cantidad de símbolos leídos desde una muestra cualquiera.

2.2. Fuente de Markov

Sea $F=\{s_1,s_2,...,s_n\}$ una fuente de símbolos de un alfabeto; y sea una sucesión de símbolos cualquiera de ese alfabeto, se define una fuente de Markov [HAM80], de la siguiente manera:

Definición 3: Se define como peso p del símbolo s_i, y se simboliza $p(s_i/s_{j1}\, s_{j2}...s_{jm})$, para un proceso de Markov de orden m a la cantidad de veces que s_i se repite en la sucesión, después de la secuencia $s_{j1}\, s_{j2}...s_{jm}$.

En este caso se dice que F es una fuente de Markov de memoria no nula, o proceso de Markov de orden m.

Para el ejemplo 2.1, sea $m = 2$, donde deben considerarse los 2 símbolos anteriores para determinar el peso de el actual, el peso de r luego de la secuencia br es $p(r/ab) = 2$.

Ahora se verá que sucede con la longitud de un código cuando se codifica una fuente de Markov de orden m.

Definición 4: Sea $F=\{s_1,s_2,...,s_n\}$ una fuente de Markov de orden m, $p(s_i/s_{j1}\, s_{j2}...s_{jm})$ el peso del símbolo s_i dado que ocurrió la secuencia $s_{j1}\, s_{j2}...s_{jm}$ y $l_i/s_{j1}\, s_{j2}...s_{jm}$ la longitud del código del símbolo s_i dado que ocurrió la secuencia $s_{j1}\, s_{j2}...s_{jm}$; la Longitud Promedio de un código para una fuente de Markov de orden m, que se simboliza L_M, se define como:

$$L_M = \sum_{j1}...\sum_{jm} \frac{1}{\Omega_{j1...jm}} \sum_i p_i l_i / s_{j1} s_{j2}...s_{jm} \qquad 2.2$$

donde $\Omega_{j1\ldots jm}$ representa el espacio muestral de la secuencia $s_{j1}\, s_{j2}\ldots s_{jm}$, es decir la cantidad de veces que ocurre tal secuencia en un conjunto de símbolos leídos.

De las fórmulas 2.1 y 2.2 se deduce una relación muy importante para esta propuesta, esta es:

$$L_{M0} \geq L_{M1} \geq L_{M2} \geq \ldots \geq L_{Mm} \qquad 2.3$$

por otro lado, en la medida que *m* crece, L_M se hace mucho menor para fuentes determinadas. Por ejemplo, en el lenguaje español es común la secuencia *"para"*, no así *"kdha"*, lo cual determina que un texto en español es un buen candidato a ser codificado como una fuente de Markov.

3. Codificación de una Fuente

3.1. Métodos Estáticos

Son varios los métodos de codificación estática, entre los más importantes pueden mencionarse Huffman [ORT94], Fano [STO88] y Shannon [HAM80]. La eficiencia de ellos se basa en dos razones fundamentales. Primero, la generación de un código lo más compacto posible, es decir, cuanto se acerca L (longitud promedio del código) a H (Información promedio de la fuente) con $H \leq L$[1]. Considerando L_H, L_F y L_{SH} como L correspondiente a los algoritmos mencionados se cumple la relación:

$$L_H \leq L_F \leq L_{SH} \qquad 3.1$$

Segundo, la simplicidad del algoritmo puede aumentar la velocidad de ejecución. Además, hay otro aspecto de suma importancia que es la cantidad de información que se necesita almacenar por cada símbolo en cada uno de los códigos, como se verá más adelante.

3.2. Método de Fano dinámico

El método elegido es el de Fano [STO88], por cuanto es veloz, y simple de implementar, que si bien, la longitud obtenida es mayor que la que se obtiene por el método de Huffman esta muy cerca de esta [ABR63].

El algoritmo que se propone es similar al que se plantea en [STO88] al que se le ha incluido un pequeño refinamiento y además se ha tomado como fuente de información de orden 1 lo que implica mantener una tabla de pesos de doble entrada, esto es que para cada símbolo anterior se tiene una tabla de pesos de los símbolos actuales. En este caso se plantea para un alfabeto código binario. El mismo puede plantearse para bases superiores, dividiendo la tabla en tantas como indique la base en la que se esté trabajando.

---

[1] *H* es la cota inferior de *L* para lograr una compresion sin perdida [HAM 61]

.2.1. Fuente de orden m

El algoritmo para codificar una fuente de Markov de orden *m* en forma dinámica por el método de Fano requiere mantener en línea una gran cantidad de información acerca de los símbolos de la fuente, tales como:

- Una tabla *n-dimensional P* de pesos, donde la dimensión de la tabla esta dada por $m+1$.
- El peso de un símbolo s_i , esta dado por $P[i,j_1, \ldots,j_m]$, es decir la cantidad de veces que ocurrió s_i luego de la secuencia $s_{j1}s_{j2}\ldots s_{jm}$.
- La tabla de pesos para cada símbolo actual es la columna *i* de *P*, esto es, el peso de cada símbolo luego de la secuencia $s_{j1}s_{j2}\ldots s_{jm}$.

3.2.2. Codificación para orden 1

Sea *P* una tabla de doble entrada en la que cada columna $P[s_i]$ es una tabla que representa los pesos (ordenados en forma descendente) de los distintos símbolos dado que ocurrió el símbolo s_i. Cada s_j es el nuevo símbolo ingresa, *n* la cantidad de símbolos distintos, l_1 y l_2 los limites de $P[s_i]$.

Puede observarse que el algoritmo 3.1 es bastante simple, además es posible implementarlo en forma recursiva ya que la naturaleza del mismo lo permite. Otro aspecto importante es que solo basta con tener una tabla de doble entrada donde cada columna contenga el carácter y su peso correspondiente, cada una ordenada en forma descendente por peso. Esta ultima observación es muy importante para una fuente de Markov de orden *m*, pues para un *m* grande es necesario mantener en memoria gran cantidad de información, como se verá más adelante.

1. $P[s_i][s_j] \leftarrow 0 \quad \forall i = 0..n$
2. $l_1 \leftarrow 0$, $l_2 \leftarrow n$
3. Si $l_1 < l_2$ → dividir $P[s_i]$ en 2 tablas (P_1 y P_2) lo más iguales posibles en peso.
   sino ir al paso 6.
4. Si *s* esta en P_1 → enviar *0* a la salida
   $l_2 \leftarrow$ limite superior de P_1
   $P[s_i] \leftarrow P_1$
   sino enviar *1* a la salida
   $l_1 \leftarrow$ limite inferior de P_2
   $P[s_i] \leftarrow P_2$.
5. Volver al paso 3.
6. Cambiar s_j por el de más arriba de la tabla cuyo peso sea menor a este.
7. Terminar.

Algoritmo 3.1 Codificación Fano dinámico en una fuente de orden 1

3.2.3. Decodificación para orden 1

El algoritmo para la decodificación es bastante similar al de codificación. La diferencia más notable entre ellos es que en el codificador se reciben símbolos de la fuente y en el decodificador del alfabeto código (por ejemplo *0* o *1*); esta diferencia puede notarse en el paso 4; otra diferencia es que se le suma el paso 6, que implica la emisión a la salida del símbolo de la fuente original.

1. $P[s_i][s_j] \leftarrow 0 \quad \forall i = 0..n$
2. $l_1 \leftarrow 0, l_2 \leftarrow n$
3. Si $l_1 < l_2 \rightarrow$ dividir $P[s_i]$ en 2 tablas (P_1 y P_2) lo más iguales posibles en peso.
   sino ir al paso 6.
4. Si es *0* $\rightarrow$ $l_2 \leftarrow$ limite superior de P_1
   $P[s_i] \leftarrow P_1$
   sino $l_1 \leftarrow$ limite inferior de P_2
   $P[s_i] \leftarrow P_2$.
5. Volver al paso 3.
6. Enviar s a la salida.
7. Cambiar s_i por el de más arriba cuyo peso sea menor a este.
8. Terminar.

Algoritmo 3.2 Decodificación Fano dinámico en una fuente de orden 1

4. IMPLEMENTACIÓN

4.1. Implementación en orden 1

Se ha implementado en C^{++} el método de Fano en forma dinámica para una fuente de Markov de orden 1 planteado en la sección anterior. La clase de la tabla de pesos *P* tiene como atributos el carácter y el peso en 2 bytes.

```
class FanoCharacter {
protected:
        char Character;
        unsigned int Occur;
public:
        FanoCharacter() : Character(0),Occur(1){};
        void SetChar(char character) {Character= character;}
        char GetChar() {return Character;}
        unsigned int GetOccur() {return Occur;}
        void AddOccur() {Occur++;}
};
```

Es necesario mantener una tabla *P* para cada símbolo s_i luego de haber ocurrido cada una de la secuencias de longitud *m* (para este caso 1), de la fuente de Markov. Para ello se declara un arreglo bidimensional de objetos, y la entrada para cada una de las 256 tablas de pesos.

*FanoCharacter huge **FanoChar = new FanoCharacter huge *[256L];*

Cada símbolo s_i nuevo involucra la necesidad de crear un nuevo objeto en la tabla correspondiente a la secuencia anterior.

FanoChar[s_i] = new huge FanoCharacter[1];

Cada una de las 256 secuencias de Markov de longitud m=1 necesita de una tabla de 256 objetos a lo sumo. Además, cada objeto ocupa 3 bytes de memoria, lo que suma un total de:

$$3 * 256^{m+1} = 3 * 256^{2} = 196608 \text{ bytes} \qquad 4.1$$

4.2. Prueba del algoritmo

Se han realizado distintas pruebas sobre archivos en un ambiente DOS, para lo cual se ha tomado como base distintos tipos de archivos tales como documentos, ejecutables y programas fuente. La comparación se hace con un algoritmo de Fano implementado en forma dinámica para fuentes de información de memoria nula (símbolos independientes).

Cada tabla se presenta de la siguiente manera: en la primera columna aparece el tamaño original del archivo en bytes, en la segunda el tamaño del comprimido mediante la implementación del método propuesto para Markov de orden 1, y la tercer columna el tamaño del comprimido mediante un método de Fano pero tomando como símbolos independientes (orden 0). La cuarta y quinta columna expresan el porcentaje que representa el comprimido del original, para Markov de orden 1 y 0 respectivamente; y la sexta columna la diferencia entre porcentajes de orden 0 menos el de orden 1.

4.2.1. Archivos Ejecutables (*.EXE)

Original	Orden 1	Orden 0	% Orden 1	% Orden 0	Diferencia
57,344	35,799	43,347	62%	76%	13%
102,335	61,911	76,875	60%	75%	15%
256,192	164,802	209,113	64%	82%	17%
518,672	277,271	372,072	53%	72%	18%
				Promedio	16%

Puede observarse que las diferencias entre ambos métodos es mayor en cuanto mayor es el archivo. esto se debe a que el método, en la medida en que avanza en la codificación, va corrigiendo las probabilidades de ocurrencia de cada uno de los símbolos.

4.2.2. Fuentes de C++ (*.CPP)

Original	Orden 1	Orden 0	% Orden 1	% Orden 0	Diferencia
25735	9619	12305	37%	48%	10%
50623	17288	23484	34%	46%	12%
113697	52276	71050	46%	62%	17%
193719	83621	128245	43%	66%	23%
				Promedio	16%

Aquí las diferencias entre ambos es mayor que en el caso de los ejecutables si se consideran archivos de gran tamaño. Además, en la medida que el tamaño del archivo aumenta la compresión es mucho mayor.

4.2.3. Documentos Word para Windows (*.DOC)

Original	Orden 1	Orden 0	% Orden 1	% Orden 0	Diferencia
21.504	9.525	12.250	44%	57%	13%
27.136	14.841	17.896	55%	66%	11%
47.104	23.702	32.051	50%	68%	18%
76.288	36.536	51.618	48%	68%	20%
131.072	61.345	88.816	47%	68%	21%
				Promedio	16%

En este tipo de archivos la diferencia se mantiene de la misma forma que para los casos anteriores. Los archivos utilizados para esta prueba contienen solamente texto (sin gráficos, tablas, etc).

Es evidente que los resultados obtenidos en la forma propuesta son mucho mejores que los de un proceso de orden 0, todas las diferencias de la sexta columna son positivas.

4.3. Consideraciones para ordenes superiores

La suma anterior no es una cantidad problemática de memoria para cualquier equipo de computación de hoy en día. Sin embargo, para una fuente de orden m=2 se necesitan 50331648 bytes (4.1), suma que si es problemática, si bien, haciendo un buen uso de la memoria, es decir, creando o destruyendo nuevos objetos es posible manejarse dentro de los limites actuales teniendo en cuenta que con un orden 2 los grados de compresión que se alcanzan son más que interesantes.

. Conclusiones

sta propuesta combina dos teorías muy importantes. Una de ellas es la codificación de Fano námico [STO88], y por el otro que este método sea tratado como una fuente de Markov de rden *m* [HAM80]. Esto tiene ventajas muy importantes entre las que se mencionan:

No existe la necesidad de relevar las probabilidades de ocurrencia de los símbolos por cuanto es un proceso inmediato en la medida que se codifica.

No es necesario mantener probabilidades de ocurrencia arbitrarias (por ejemplo históricas) con lo que se reduce el tamaño del algoritmo o información en archivos adicionales.

- En la compresión de archivos no hace falta grabar en la cabecera del mismo los símbolos con sus códigos, que para fuentes de Markov con *m* grandes, aumentaría enormemente el tamaño de los archivos comprimidos.
- En archivos más grandes la relación de compresión suele ser mejor, y esta es tal vez la conclusión más importante, por cuanto el ahorro de almacenamiento es mucho mayor.

6. Trabajos Futuros

- Implementación del método para *m* superiores a 1, con esto se reduciría notablemente la Longitud Promedio del código.
- Otros métodos en forma dinámica pueden implementarse, Huffman por ejemplo [RUE95], que si bien necesitan mayor cantidad de memoria reducen la Longitud Promedio del código.

7. Bibliografía

[ABR63] Abramson N., Information Theory and Coding, McGraw-Hill, 1963.

[HAM80] Hamming Richard, Coding and Information Theory, Prentice Hall, 1980.

[LIP88] Lipschutz S., Estructuras de Datos, Mc Graw Hill, 1988.

[ORT94] Ortega M. - Klenzi R. - Rodriguez N. - Rueda L., "Software Compactador usando estrategia Huffman de Compactación", II Congreso y Exposición Internacional de Informática, Mendoza, Argentina, 1994.

[RUE95] Rueda L. - Ortega M. - Klenzi R. - Rodriguez N., "Un modelo de Codificación Dinámica del método de Huffman para la Compresión de Datos", 2° Congreso Argentino en Ciencias de la Computación, Bahía Blanca, Argentina, 1995.

[STO88] Storer J., Data Compression: Methods and Theory, Computer Science Press, 1988.

[VAN86] Van Lint, Introduction to Coding Theory, J. GMT 86.