Tarea Programada

Codificación Huffman

Descripción:

La codificación Huffman es un algoritmo de compresión sin pérdida (lossless compression) de datos inventado por David Huffman en sus años de estudiante (inventó el algoritmo para su curso de *Information Theory* del MIT). Este algoritmo aprovecha la aparición repetitiva de los valores que forman un documento para crear una codificación de tamaño variable que permita almacenar la misma información en un espacio menor.

Un ejemplo de esto sería el caso de los documentos de texto en donde se hace un amplio uso de caracteres alfanuméricos, pero el resto de caracteres de la tabla ASCII (incluyendo caracteres especiales) casi no se utilizan. De esta manera el algoritmo genera una codificación en la que los caracteres más utilizados tienen una representación más corta (e.g: 4-6 bits por caracter) y los caracteres menos utilizados tienen una representación más larga (e.g: 10-12 bits). De esta manera, aunque ciertos caracteres ahora requieren más espacio, en promedio la cantidad de bits/caracter disminuye, lo que permite "comprimir" el archivo.

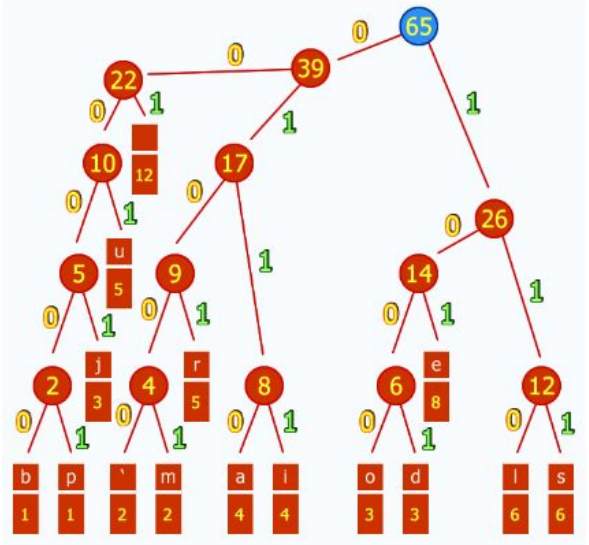
El algoritmo consiste en 3 etapas principales: El conteo, la construcción de la codificación y la transcripción.

1. En la primera etapa se listan todos los caracteres que componen el documento a comprimir y se cuenta cuántas veces aparece cada uno de ellos.

2. En la segunda etapa se construirá un árbol binario que representará la codificación a utilizar. Los caracteres leídos del documento y sus conteos respectivos se convierten en nodos de árbol binario (sin estar vinculados a ningún otro nodo: son nodos sueltos) y se introducen en una cola de prioridad, donde la prioridad está dada por el conteo, dando prioridad a los valores menores. Posteriormente se ejecutarán las siguientes instrucciones hasta que solo quede un elemento en la cola:
    a. Se sacan de la cola los dos siguientes elementos.
    b. Se crea un nuevo nodo de árbol binario, cuyos hijos son los dos elementos extraídos.
    c. El conteo del nuevo nodo es la suma de los conteos de sus dos hijos.
    d. El nuevo nodo se inserta a la cola de prioridad.
    e. Este proceso se repite hasta que solo quede un elemento en la cola.
    f. Cuando solo queda un elemento, ese elemento es la raíz del árbol de codificación.

Ejemplo de construcción del árbol de codificación

3. En la tercera etapa se transcribe el documento: para cada caracter del documento se encuentra su camino dentro del árbol de codificación. Por convención cada hijo izquierdo se considera una 0 y cada hijo derecho un 1. De esta manera en el árbol de ejemplo anterior la codificación binaria del caracter 'u' sería representada por un: 0001 en lugar del ASCII 01110101.



Es importante al utilizar este tipo de codificación adjuntar la información con respecto al documento original y el árbol de codificación, de manera que el archivo pueda ser decodificado con facilidad. Para ello se puede almacenar al inicio del documento el tamaño original del documento para que al realizar la decodificación del archivo comprimido se sepa

de antemano el espacio que ocupará. También se almacena la cantidad de bits "sobrantes" de la codificación [0-7], esto porque al convertir los bits en bytes, pueden quedar bits que no alcanzan para un byte completo. Además, se puede almacenar la cantidad de símbolos utilizados y luego para cada símbolo utilizado, el símbolo ASCII y su conteo respectivo para reconstruir el árbol. En este caso es necesario que los elementos sean ingresados a la cola de prioridad en el mismo orden en el que se ingresaron al codificar.

Al momento de decodificar se realiza un proceso similar al de codificación:

1. El documento codificado incluye información con respecto al árbol de codificación. El primer paso es utilizar dicha información para reconstruir el árbol de manera que quede idéntico al árbol original.

2. Con el árbol de codificación es posible realizar la decodificación del documento siguiendo el siguiente algoritmo:
    a. Se inicia en el nodo raíz del árbol y se lee el primer bit del documento.
    b. Dependiendo del bit leído se avanza al siguiente nodo hijo izquierdo o derecho.
    c. Se continúan leyendo bits y descendiendo por el árbol hasta llegar a un nodo hoja (que no tiene hijos).
    d. Al llegar a un nodo hoja se toma el símbolo del nodo y se escribe en el documento decodificado.
    e. Se regresa al nodo raíz del árbol, se lee el siguiente bit del documento y se regresa al punto b.
    f. Cuando no quedan más bits por leer se finaliza el proceso.

Tarea programada: **Es MUY importante que tenga un 100 y todos los puntos extra dado que me puede subir mucho la nota.**

En esta tarea el objetivo consiste en implementar de manera adecuada el código necesario para la ejecución de la codificación y decodificación de un arreglo de caracteres por medio del algoritmo de codificación de Huffman.

Para ello usted deberá implementar las siguientes estructuras y funciones (puede utilizar otras estructuras/funciones de soporte o implementar su propia versión de las estructuras/funciones mientras el resultado obtenido sea el mismo):

1. (5%) Estructura `Symbol`: Una estructura que posee los siguientes atributos: un caracter *symbol* que almacena el caracter representado y un entero positivo *count* que permite contar cuantas veces aparece el símbolo.

2. (10%) Estructura `BinaryNode`: Una estructura que permite almacenar un caracter *symbol*, un entero positivo *count* y punteros a un hijo izquierdo *left* y un hijo derecho *right*.

3. (5%) Estructura `CodingTree`: Una estructura árbol que posee por raíz un puntero a un BinaryNode y posee los siguientes métodos para recorrer el árbol:

a. (10%) `bool encode(char* code,char c)`: Calcula la codificación de 1's y 0's necesaria para alcanzar el nodo con caracter *c* y almacena dicha codificación como una hilera de '1's y '0's en el arreglo *code*. Retorna si se logró encontrar la codificación.

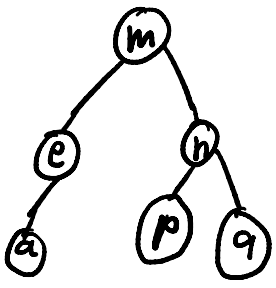
b. (10%) `char decode(const char* code,unsigned int &read)`: Recibe una hilera *code* con una secuencia de '1's y '0's y recorre el árbol para hasta alcanzar un nodo hoja y retorna el caracter asociado a la secuencia. En la variable *read* anota la cantidad de "bits" (caracteres) leídos del arreglo.

c. (5%) `unsigned int count()`: Calcula la suma de los contadores de todos los nodos no-hoja (nodos que tienen hijos). Este valor es la cantidad de bits que ocupará el archivo codificado. O en el caso de una hilera de '1's y '0's es la cantidad de bytes que ocupará dicha hilera.

4. (10%) Estructura `PriorityQueue`: Una estructura cola de prioridad que almacena punteros a BinaryNodes, la prioridad está dada por el conteo de cada BinaryNode, con prioridad a números menores. Debe poseer los métodos `push`, `pop` y `getSize`.

5. (10%) Escriba un método `CodingTree* createCode(Symbol** symbols)`: Este método recibe por parámetro un arreglo de punteros a objetos Symbol *symbols* y retorna un puntero a un objeto CodingTree con la codificación que se obtiene para dichos símbolos.
   a. Los objetos Symbol ya vienen con su caracter y conteo respectivos.
   b. Primero creará objetos BinaryNode de los objetos del arreglo *symbols* y los agregará a una cola de prioridad. Solo tome en cuenta a aquellos Symbol cuyo conteo sea mayor a 0.
   c. Luego utilizará la cola de prioridad para ejecutar el algoritmo de construcción del árbol de codificación (*CodingTree*).
   d. Retorne el árbol de codificación creado.

6. (15%) Escriba un método `char* encode(const char* str, unsigned int length, Symbol** symbols, unsigned int &lengthOutput)`: Este método recibe por parámetros un arreglo de caracteres *str* que se desea codificar, un entero positivo *length* con el tamaño del arreglo a codificar, un arreglo de punteros a objetos Symbol *symbols* y un entero positivo referenciado *lengthOutput*. El método retorna un arreglo de caracteres con el resultado de la codificación de *str*. Además se almacena el tamaño del arreglo retornado en la variable *lengthOutput*.
   a. Primero reiniciará el valor de los contadores de cada objeto en *symbols* a 0.
   b. Luego contará la cantidad de veces que aparece cada char en *str*, actualizará el valor del contador del Símbolo en *symbols* correspondiente.
   c. Obtenga el árbol de codificación usando `createEncoding(symbols);`
   d. Cree un nuevo arreglo para almacenar el resultado de la transcripción. El tamaño de dicho arreglo se puede calcular utilizando el método count del árbol de codificación y se almacena en *lengthOutput*.
   e. Utilizando el árbol de codificación generado se procederá a transcribir el arreglo de caracteres *str* en su versión codificada en el arreglo *result*.



Es muy importante que desde los primeros métodos le vayan saliendo bien para que tenga toda la tarea buena. Haga una buena organización de como va a trabajar.

f. Puntos extra y puntos de honor +10: Convierta la hilera de ‘1’s y ‘0’s a su codificación binaria, de manera que el arreglo *result* utilice solo ⅛ del espacio de almacenaje. Para ello, lea la documentación de los *bitwise operators* de C/C++: ~ & | ^ << >>

g. Se retorna el arreglo *result*.

7. (15%) Escriba un método `void decode(const char* encoded, unsigned int length, Symbol** symbols, char* str)`: Este método recibe por parámetros un arreglo de caracteres *encoded* que se desea decodificar, un entero positivo *length* con el tamaño del código almacenado en *encoded*, un arreglo de punteros a objetos Symbol *symbols* con la información de los conteos de caracteres y un arreglo *str* que se utilizará para almacenar el archivo decodificado.

   -7pts ✓

   a. Primero obtenga el árbol de codificación usando `createEncoding(symbols);`
   b. Puntos extra y puntos de honor +10: Si realizó los puntos extra 6.f entonces el arreglo *encoded* no poseerá una hilera de ‘1’s y ‘0’s, sino que los datos vendrán verdaderamente comprimidos. Cree una nueva hilera con el resultado de “descomprimir” *encode* en una hilera de ‘1’s y ‘0’s.
   c. Utilice su árbol de codificación para decodificar la hilera de ‘1’s y ‘0’s, guarde los resultados de las decodificación en *str*.

8. (5%) Implemente un código *main* que deberá hacer lo siguiente:

   -3pts

   a. Crea un arreglo *symbols* de 256 punteros a objetos Symbol, debe inicializar cada objeto con un caracter diferente para representar los 256 posibles caracteres ASCII.

```
char t[] = “All the world's a stage, and all the men and women merely players.”;
```

   b. Haga un llamado a `char* code = encode(t,67,symbols, compressedLength)`. Si su código ejecuta correctamente debería producir el resultado:

```
10111001001011000100011101111010011100100000010111101010110001
11011101100001001011101011110111011001101110100011110110111001
00101100010001110111101111110111000110111010001111011010011 1001
01111101110001101111101100000110101010011000110010111010100011
0000000110110110101 0
```

   Si realiza la compresión a nivel de bytes el resultado sería algo similar a:

   ╣║,Gzr♣Ω║╟v↕δ▌ ¢ú█Æ—w╛ πt{N_q╛ ┴¬c.ú☺╢║á

   c. Cree un arreglo de caracteres *res* de tamaño 67. Luego haga un llamado a `decode(code, outputLength, symbols, res)`. Si su código ejecuta correctamente debería producir de resultado la hilera original.

d. Recuerde eliminar debidamente **todos** los objetos creados durante la ejecución de su código. No deben haber fugas de memoria.

Otro caso de prueba:

```
char v[] = “But on this most auspicious of nights, permit me
then, in lieu of the more commonplace soubriquet, to suggest
the character of this dramatis persona. Voila! In view humble
vaudevillian veteran, cast vicariously as both victim and
villain by the vicissitudes of fate. This visage, no mere
veneer of vanity, is a vestige of the vox populi now vacant,
vanished. However, this valorous visitation of a bygone
vexation stands vivified, and has vowed to vanquish these
venal and virulent vermin, van guarding vice and vouchsafing
the violently vicious and voracious violation of volition. The
only verdict is vengeance; a vendetta, held as a votive not in
vain, for the value and veracity of such shall one day
vindicate the vigilant and the virtuous. Verily this
vichyssoise of verbiage veers most verbose, so let me simply
add that it is my very good honour to meet you and you may
call me V.”;
```

Produce de resultado:

```
00101000101101100011110110100111100011000110100111111100111011
00111000111101001101001110011011101001001101101101101001111110
11011111111010011010111101100010000011001011111100110100001110
11001111011000111110011000111100011000000010000101111111010100
11110010110100001101111101101111111110001100000011111001110110
11100001110010010111100111100111011010010011011001… (etc.)
```

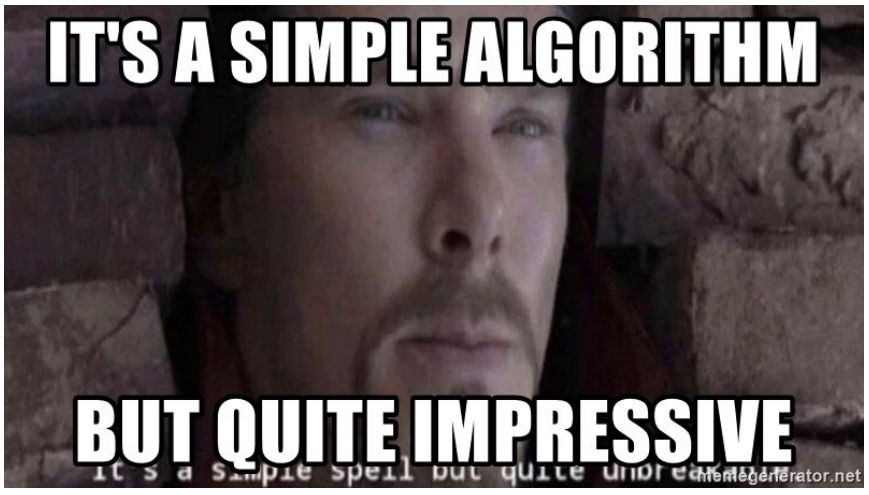
```
(┤=ºîiⁿ∞π└Nnômº∅■Ü÷≤C│┼·Å_└≥┴▀o°└·▉ç%τ¥ñ┘Q;l├l┘ó♂±│┌≤└
π├&)└H♫÷îi÷:│¿└≤CÄ┬è·┬┘T┴i┼D∩├sÖ!┌Ü└┐)m'Q☺└à≥Q├]%;mº,┘▼ÖqÄ
│H▄²#:∞ÑZ₧f^0:Θ4┴a¼∅■²@+τÿ└δºO☻·_├8♥┘┐┌ôc"■ƒ«áπ^▲▀±ü┌τ
▌m─l┐Ñ┼┘óRA}i4≡↑Wö«t(q~1º┌Ñn┤ƒ]=èì┤÷»3'¡Q>¿▉NqH├δ½»Φa~æƒ¶}
nt3┼┘ñ₧┐O▲0♀uR⌠îδ«léGP∞┘♂δI▌m9ÜÅu┬☼H╪┤┬`Θ²G┘`u▉ÉHû]t¢m?H╪
┤Ω&▉O«▄¬6└▉²n[←hW┴0=ñ▒Ω╙─$}>í▲¶ä♣☼]B0DQ⌂☻L⌠~│▄jÄù▼S¡j↨▀┐x└
δRh⌠îΩ↔D┼┬┤Ö_8↔▒┬├4·Θ1æ¥ΘÅo┘♫Ö┤ ≡Ω♥Åτg↔Cªyé·▀ÉÅÿτ┤─û^ÿ┴↑¿
√▼Oτùp∞┐┼l°┤ ┤┐┼ⁿ └Åò┐┬3σm·┌\ò)│├£+?
```

Tips de desarrollo y debuggeo:

1. Esta es posiblemente la primera vez que usted se encuentra ante un desafío tan “grande” (~250 LOC). Sin embargo, durante los años siguientes notará el tamaño incremental de los desafíos y programas a desarrollar (en los cursos de ingeniería de software sus programas se medirán en miles de líneas de

código posiblemente). Sin embargo, el primer secreto es no dejar que la cantidad de trabajo o el tamaño del problema lo abrume.

2. La siguiente etapa es realizar un "divide y vencerás". Todo problema puede ser descompuesto en problemas más pequeños, en este caso su profesor le ofrece una manera de visualizar el problema dividido ("en mis tiempos solo nos entregaron la descripción del algoritmo y a cada quién le tocó descubrir cómo resolverlo"), pero usted puede desarrollar su propia implementación. En caso de estar trabajando en grupo también es útil hacer un análisis de dependencias: "¿cuáles componentes requieren de cuáles otros?", para ver cuáles secciones del problema se pueden desarrollar en paralelo y como dividir el desarrollo.
3. Aunque parecen ser muchos componentes, el problema se vuelve más fácil si uno realiza las pruebas por partes: pruebe las clases de manera independiente antes de unir todo su código, es más difícil (más no imposible) debuggear un código corriéndolo una vez terminado.
4. Los "cout"s y "printf"s son sus amigos. Les permiten saber información con respecto al código en tiempo de ejecución. Por ejemplo: puede imprimir los valores de conteo de cada Symbol después de realizar el conteo y ver si el código va funcionando bien "hasta ahí". De la misma manera si su programa *crashea*, poner prints le puede ayudar a averiguar en cuál línea de código es donde el programa está fallando. Siga el camino de migajas.
5. Si su programa da errores de compilación, en la sección de mensajes es posible ver el número de línea del error (usualmente se presenta como Directorio/NombreDelArchivo.cpp:#, donde # es el número de línea). Sabiendo el número de línea y leyendo la descripción del error es más fácil solucionar los errores.
6. A la hora de manipular los arreglos e hileras, no tema usar punteros. Recuerde que es posible tener acceso de `char*` en una posición de una hilera usando una notación de acceso y referencia: `&arreglo[index];`
7. Finalmente, si desarrolló su código utilizando al menos la clase Symbol entonces podrá utilizar la biblioteca proveída por su profesor en este link para cargar y guardar archivos tanto regulares como codificados.



Puede utilizar el código del profesor para probar su compresor cifrando por ejemplo un libro en formato .txt (en mi ejemplo utilizo el primer libro de Harry Potter) con un código similar al siguiente:

```
Symbol** symbols = new Symbol*[256];
for(int i=0;i<256;i++){ symbols[i] = new Symbol((char)i); }
char *s,*code;
unsigned int originalLength,compressedLength;

// Ejemplo de compresion
s = loadFile("HP1.txt",originalLength);
code = encode(s,originalLength,symbols,compressedLength);
saveCompressedFile("HP1.cde",code,compressedLength,originalLength,symbols,
true);
delete[] s;
delete[] code;
originalLength = 0; compressedLength = 0;

// Ejemplo de descompresion
code =
loadCompressedFile("HP1.cde",compressedLength,originalLength,symbols,true)
;
s = new char[originalLength];
decode(code, compressedLength, symbols, s);
saveFile("HP1_decoded.txt",s,originalLength);
delete[] s;
delete[] code;

for(int i=0;i<256;i++){ delete symbols[i]; }
delete[] symbols;
printf("Press ENTER...");
fgetc(stdin);
return 0;
```

Ambos archivos (HP1.txt y HP1_decoded.txt) deben ser idénticos y del mismo tamaño. Además es posible notar que, si se utiliza compresión a nivel de bytes: mientras que el libro original está compuesto por 448810 bytes, el archivo comprimido (HP1.cde) está compuesto por tan solo 260121 bytes, lo que representa una compresión del 43%.

Por motivos de los tipos de datos utilizados solo es recomendable comprimir archivos inferiores a los 500 Mbs.