Reporte tarea 3 - Programación y algoritmos

Erick Salvador Alvarez Valencia

Centro de Investigación en Matemáticas, A.C.,
erick.alvarez@cimat.mx

Resumen En el presente reporte se analizará la elaboración de los programas asignados para la tarea no. 3 de la materia de programación y algoritmos, así como una pequeña descripción de las ejecuciones realizadas y los resultados producidos por las mismas. Y al final se añadirán conclusiones propias las cuales incluyen algunas mejoras hacia los algoritmos.

Keywords: Lenguaje C, punteros, memoria dinámica.

1. Preguntas sobre arreglos

1. **Hemos comentado que es mas eficiente que una matrız sea un solo bloque de memoria. ¿Cómo podrıamos lograr eso con memoria dinámica (y accediendo con doble corchete $x[i][j]$).**
   Una manera de hacer esto es creando un puntero de punteros, pero a diferencia de como se vió en clase, en la primera posición de este arreglo se reserva todo el bloque de memoria, el cuál será de $nr * nc$. Así mismo esta posición fungirá como apuntador para el comienzo de toda la memoria que se reservó.
   Las otras $n - 1$ posiciones del arreglo de punteros debemos referenciarlas al comienzo de cada fila de la matríz (de esta forma lo podemos interpretar), lo cual podemos hacer con un ciclo.

2. **¿Cómo requerimos memoria para una matriz triangular?**
   La idea es construir una matríz por bloques, primero reservar memoria para el número de filas usando el malloc con doble apuntador, y posteriormente en un for pedir memoria para las columnas, sólo que en esta parte el tamaño de las columnas debe ir incrementando uno con respecto al anterior, de forma que para la primera columna reservemos sólo un espacio, para la segunda dos, y así sucesivamente. Podemos apoyarnos de la variable del for externo para realizar esto.

3. **¿Cómo podríamos requerir memoria para que los índices comenzaran desde 1 en lugar de 0?**
   Se reserva la memoria normalmente con malloc para n posiciones, y al final solo tenemos que mover nuestro apuntador una posición hacia la izquierda, de esta forma todas las peticiones que se hagan hacia una determinada posición del arreglo, se deben de hacer uno más hacia la derecha, ya que al recorrernos a la izquierda una vez, la posición uno se convierte en cero, la dos en uno y así.

4. **¿Cómo podríamos requerir una locacion de memoria extra para guardar el tamano de un bloque?, por ejemplo, para funciones que solo reciben el apuntador al bloque de memoria, y en la posición -1 tiene el tamano del arreglo.**
   Es la misma estrategia que para la cuestión anterior, solamente que debemos reservar un bloque más de memoria y avanzar el punterio una posición hacia adelante para crear el efecto contrario, de esta forma podemos hacer que $arr[-1]$ sea una posición válida para almacenar el tamaño.

2. Programa 1 - Alocación de memoria

2.1. Descripción

Para este programa se crearon funciones, las cuales generaban y eliminaban vectores y matrices con memoria dinámica. Para las matrices de dos dimensiones se creó una funcion especial, en la cual se verificaba cuanta memoria

Algorithm 1 Creación de una matríz continua o por bloques

```
1: procedure CreateMatrix2D(nc, nr, typeSize)
2:     MB ← 10 * 1000 * 1000
3:     if nc * nr * typeSize < MB then
4:         Crear matríz continua.
5:     else
6:         Crear matríz por bloques.
```

se solicitaba, si dicha memoria era menor que 10 MB se creaba la matríz en forma contínua (la memoria de filas y columnas se almacenaba en un solo vector) o por bloques.

A continuación se despliega el pseudocódigo de la creación de las matrices continua y por bloques. La función de creación de un vector se omite ya que prácticamente viene contenida en las presentes.

Algorithm 2 Creación de una matríz continua

```
1: procedure CreateMatrix2DNoChunked(nc, nr, typeSize)
2:     Mat ← puntero de punteros tipo void.
3:     Se reserva memoria con malloc para el número de filas de la matríz.
4:     for i = 1 to nr do
5:         Mat[i] = Mat[i − 1] + nc * typeSize
6:     return Mat
```

Algorithm 3 Creación de una matríz por bloques

```
1: procedure CreateMatrix2DChunked(nc, nr, typeSize)
2:     Mat ← puntero de punteros tipo void.
3:     Se reserva memoria con malloc para el número de filas de la matríz.
4:     for i = 1 to nr do
5:         Mat[i] = malloc(nc * typeSize)
6:     return Mat
```

2.2. Descripción de las ejecuciones

Para probar el programa, este se ejecutó de manera que se solicitaban diferentes tamaños de memoria, para analizar si la matríz se generaba correctamente, después se usó la herramienta valgrind para verificar si existía memoria no liberada al finalizar la ejecución del mismo, de lo cual se analizó que todo funcionaba en orden.

3. Programa 2 - Análisis de frecuencias de un conjunto de textos

3.1. Descripción

Para este programa, se tomaba un conjunto de archivos, los cuales fueron generados en la tarea anterior, posteriormente se tenía que agrupar sus palabras en una matríz de tal forma que se fueran guardando las palabras que eran distintas, pero de igual manera se tenía que almacenar su frecuencia, para hacer esto se podía usar un vector de enteros.
La parte crucial era el hecho de que las palabras y las frecuencias se tenían que ir almacenando en una estructura dinámica que crecía conforme a como se iban agregando las mismas. Cabe destacar que la matríz de caracteres

donde se almacenaron las palabras y el vector donde se almacenaron las frecuencias se crearon usando la librería anterior.

A continuación se muestra el pseudocódigo de la función principal del programa:

Algorithm 4 Procesamiento de las frecuencias relativas de un conjunto de archivos

```
1: procedure PROC
2:     Dict ← puntero de punteros tipo char.
3:     Hist ← puntero de enteros.
4:     cnt ← 0
5:     Leer El nombre del archivo.
6:     Leer La cantidad de archivos.
7:     Inicializar Dict con diez filas y sin memoria para columnas.
8:     Inicializar Hist con diez espacios.
9:     for i = 1 to no_archivos do
10:        Abrir el i-ésimo archivo.
11:        while Haya una siguiente palabra en el archivo do
12:            Leer palabra.
13:            cnt ← cnt + 1
14:            Convierte palabra a minúsculas.
15:            Añade palabra a Dict.
16:            Actualiza la frecuencia de palabra en Hist.
17:    Imprime Dict y Hist.
18:    Libera memoria.
```

3.2. Descripción de las ejecuciones

Para probar este programa, como se mencionó en la sección anterior, se usaron los archivos de la tarea anterior, se supone que al final, este programa debía generar un archivo que contuviera la lista de palabras no repetidas y su frecuencia relativa, esto se logró con la frecuencia de cada palabra dividiéndola con el número total de palabras.

Primeramente se corrió el programa con los 59 archivos generados con el primer archivo procesado en la tarea anterior, se analizó que el programa funcionó correctamente generado el archivo de las frecuencias relativas. De la misma forma se ejecutó el programa con la ayuda del software valgrind para verificar errores de fugaz de memoria.

A continuación se muestra un ejemplo de la salida producida por el programa.

```
Palabras Frecuencia relativa.
the 0.059113
project 0.001655
gutenberg 0.000571
ebook 0.000209
of 0.034255
prince 0.004222
by 0.009624
nicolo 0.000285
machiavelli 0.001027
this 0.006961
is 0.008312
for 0.008445
use 0.000571
anyone 0.000095
anywhere 0.000038
at 0.003994
no 0.002168
cost 0.000076
and 0.036784
with 0.009453
almost 0.000171
restrictions 0.000038
whatsoever 0.000038
you 0.004565
may 0.002092
copy 0.000228
it 0.011127
give 0.000590
away 0.000456
or 0.005268
reuse 0.000038
under 0.001426
terms 0.000475
license 0.000304
included 0.000095
online 0.000076
wwwgutenbergorg 0.000057
title 0.000152
author 0.000038
translator 0.000019
w 0.000057
k 0.000038
marriott 0.000038
release 0.000038
date 0.000095
february 0.000038
11 0.000019
2006 0.000019
1232 0.000019
file 0.000057
last 0.000171
updated 0.000038
october 0.000019
19 0.000019
2010 0.000019
language 0.000114
english 0.000095
```

(a) Figura 1 (Salida del programa).

Cabe destacar que en la Figura 1 no se muestran todas las palabras impresas por el programa.

4. Conclusiones generales

Queda concluir que ambos programas funcionaron según lo esperado, para el segundo programa se pudo realizar algunas mejoras como: Un algoritmo de búsqueda más optimo, como la búsqueda binaria, la cual se realiza con estructuras ordenadas, aunque eso implicaría que al añadir las palabras a la matríz se hiciera siguiendo un orden, y de la misma forma cuidar el mismo orden en las frecuencias.

Otra mejora podría haber sido el usar una matríz continua, el problema con esto, es que si crece bastante la matríz tendríamos que migrarla a una por bloques y eso de alguna forma podría ser costoso, por el tiempo de procesamiento.