|  |
| --- |
| Universidad blas pascal |
| Trabajo Final |
| Programación eficiente |
|  |
| **Cenzano Sebastian** |
| **03/02/2019** |

|  |
| --- |
|  |

# Consigna

El objetivo de este Trabajo Final es tomar un programa de alguna materia anterior, proyecto personal o descargado de Internet y analizarlo en detalle suﬁciente como para poder determinar qué correcciones, mejoras y optimizaciones se deberían aplicar. Se deberán analizar los siguientes puntos:

* Cobertura del código fuente,
* Detección de las funciones que realizan el mayor consumo de CPU,
* Redundancia de operaciones aritmético–lógicas,
* Modo de utilización de la memoria asignada en forma dinámica,
* Modo de acceso a la memoria caché y principal.

Las pruebas a realizar deberán estar sistematizadas para así tener una medición objetiva y repetible. Una vez analizado el software, se deberán proponer mejoras a realizarle. Luego, se procederá a su implementación. Finalmente, se deberá hacer un estudio comparativo entre el software original y su versión mejorada aplicando, en caso de ser posible, las mismas pruebas de software sistematizadas. En esta etapa se incluirán todas las tablas y gráﬁcos necesarios para mostrar el impacto de los cambios realizados.

# Problemática abordada

El programa a utilizar fue provisto en la materia “Algoritmos y estructuras de datos 2”. El mismo, utiliza el Algoritmo de Huffman para comprimir un archivo provisto como entrada. Además, cuenta con la opción de descomprimir el archivo utilizando el mismo Algoritmo.

En la fase de compresión, se toma un archivo binario, se analiza su contenido y se crea un árbol de Huffman cargando los nodos con la información de los bytes y su frecuencia de aparición. Luego, información como la tabla de símbolos y el vector que representa al árbol se guardan en el archivo a comprimir, terminando con el proceso de compresión.

Para descomprimir el proceso es, a grandes rasgos, el inverso.

Con el objetivo de utilizar las herramientas vistas durante el dictado de la materia, el programa se compilara, ejecutara y testeara en una distribución del sistema operativo Linux. El software a testear, originalmente programado para Windows, requirió cambios mínimos para que su uso sea el esperado, como ejemplo corregir las instrucciones “system(‘pause’)” mediantes las cuales el programa espera alguna acción de parte del usuario para seguir con la ejecución.

# Herramientas

Las herramientas a utilizar para el profiling son las siguientes:

* Gcov: Para medición de cobertura de código. Se incluirán los reportes generados por la herramienta.
* Gprof: Para obtener información sobre consumo de CPU, discriminado por función. Orden y cantidad de llamada de las funciones. Se incluirán los reportes generados.
* Valgrind: Para obtener información sobre manejo de memoria.

# Ejecución & diagnostico

Realizaremos pruebas, con las herramientas previamente mencionadas, en el software sin modificaciones para poder obtener un primer diagnostico. Luego analizaremos los resultados y en base a estos, realizaremos las mejoras vistas a través de la materia.

## Compilación

Para Compilar el software se utilizara el compilador “g++” en su versión 7.3.0 la cual utiliza por defecto el standard c++14.Realizamos una primera prueba de compilación con los siguientes parámetros:

g++ -o HuffmanCompressor.exe \*.cpp  
Y obtenemos los siguientes resultados.

object.cpp: In member function ‘virtual std::\_\_cxx11::string Object::toString()’:object.cpp:23:30: error: cast from ‘Object\*’ to ‘int’ loses precision [-fpermissive]  
 sprintf( cad, "%x", (int)this );  
 ^~~~  
object.cpp: In member function ‘virtual int Object::hashCode()’:  
object.cpp:32:17: error: cast from ‘Object\*’ to ‘int’ loses precision [-fpermissive]  
 return (int)this;  
 ^~~~

Estos errores se deben a que la implementación del software utiliza clases “Objeto” (emulando las superclases provistas por Java) que, en este caso, calculan el hash de un objeto simplemente casteándolo a un objeto del tipo int.Habiendo confirmado el correcto funcionamiento del programa, procederemos a utilizar el flag de compilación “-fpermissive” para seguir con las pruebas.

## Planificación de las pruebas

Los archivos de prueba consisten en archivos de imágenes con la extensión .jpg de tamaños variados, por lo que se creó un bash script para la compilación y el uso de cada una de las herramientas de profiling y optimización:

* Script-GCOV.sh
* Script-GPROF.sh
* Script-Valgrind.sh

## Cobertura

### Configuración

Para realizar las pruebas dinámicas de cobertura del cogido, utilizaremos la herramienta de GCC llamada GCOV. Con los resultados de estas pruebas podremos remover código que no se utiliza y entender cuáles son las áreas más utilizadas del código, y que por lo tanto, mejores resultados traerán al mejorar su performance.Como mencionamos previamente, las pruebas se normalizaron con la utilización de scripts y los pasos se describen a continuación.  
  
La compilación se realizo con los siguientes argumentos:

g++ -ftest-coverage -fprofile-arcs -fprofile-dir=./Corrida-$nombre ../../HuffmanOriginal/\*.cpp -fpermissive -o HuffmanCompressor

Los flags *–fprofile-arcs y –ftest-coverage* son mandatorios para utilizar gcov ya que, entre otras cosas, crean un grafico de flujo del programa. Por una cuestión de comodidad, se agregó también el flag *–fprofile-dir* para almacenar los archivos .gcda en un directorio separado.Para obtener realmente un indicador de cobertura del programa, el mismo se corrió tres veces en cada archivo de prueba. Una vez para comprimir el archivo, una segunda para descomprimirlo y una tercera sin argumentos para imprimir el mensaje de cómo debe utilizarse el programa.. Esto permite ejecutar todo el código “Alcanzable” del programa.

./HuffmanCompressor comprimir "$nombre.jpg" ./HuffmanCompressor descomprimir "$nombre.cmp"  
./HuffmanCompressor

Resultados coberturaEl análisis general de cobertura demuestra que se ejecuto un 66.86% de un total de 528 lineas para la compresión y descompresión de los 3 archivos.

Del informe final procederemos a analizar los números de los archivos que no han sido cubiertos al 100% para luego ir al código fuente. Obviaremos archivos que no corresponden al repositorio, tales como “iostream” ignoraremos también a efectos practicos de este trabajo los archivos “.h”.

* File '../../HuffmanOriginal/FilaDePrioridad.cpp'  
  Lines executed:3.23% of 31  
  Creating 'FilaDePrioridad.cpp.gcov'
* File '../../HuffmanOriginal/Frequency.cpp'

Lines executed:57.14% of 14

Creating 'Frequency.cpp.gcov'

* File '../../HuffmanOriginal/Heap.cpp'

Lines executed:66.95% of 118

Creating 'Heap.cpp.gcov'

* File '../../HuffmanOriginal/HuffmanCode.cpp'

Lines executed:28.57% of 21

Creating 'HuffmanCode.cpp.gcov'

* File '../../HuffmanOriginal/HuffmanCompressor.cpp'

Lines executed:80.63% of 160

Creating 'HuffmanCompressor.cpp.gcov'File

* ../../HuffmanOriginal/HuffmanNode.cpp'

Lines executed:84.21% of 19

Creating 'HuffmanNode.cpp.gcov'

* File '../../HuffmanOriginal/HuffmanTree.cpp'

Lines executed:75.95% of 79

Creating 'HuffmanTree.cpp.gcov'

* File '../../HuffmanOriginal/main.cpp'

Lines executed:100.00% of 12

Creating 'main.cpp.gcov'

* File '../../HuffmanOriginal/Nodo.cpp'

Lines executed:5.00% of 20

Creating 'Nodo.cpp.gcov'

* File '../../HuffmanOriginal/object.cpp'

Lines executed:33.33% of 15

Creating 'object.cpp.gcov'

De estos resultados podemos corroborar que ciertos archivos no están influyendo en el funcionamiento del programa. Luego de realizar todos los análisis, en la etapa de corrección realizaremos pruebas removiendo estos archivos del repositorio y comprobaremos si es que el software mantiene su funcionamiento.  
Los archivos no utilizados son los siguientes:

* FilaDePrioridad.cpp
* Nodo.cpp

El archivo “Main.cpp” que creamos para estas pruebas se encuentra entre los archivos que han sido cubiertos en un 100%. Este archivo utiliza las principales funcionalidades del software que son las de comprimir y descomprimir por lo que deberemos analizar la cobertura archivo por archivo.