**Informe De Análisis Y Diseño Del Desafió I**

Manuela Galeano Chica y Carlos Andrés Buelvas

Facultad de ingeniería, Universidad de Antioquia, Medellín

Informática II

Aníbal Guerra Soler y Augusto Salazar

25 de abril de 2025

Imagen que contiene Patrón de fondo

Descripción generada automáticamente

* **1. Introducción**

La empresa Informa2 nos solicita diseñar un algoritmo capaz de reconstruir una imagen distorsionada mediante transformaciones a nivel de bits. Para lograrlo, se nos proporcionan:

* Imágenes BMP distorsionadas.
* Archivos .txt con los resultados de enmascaramientos.
* Una imagen de ruido (IM).
* Una máscara (M) de tamaño menor.

Este informe presenta el análisis de las transformaciones aplicables, su reversibilidad y la propuesta de estructura para revertir la imagen a su estado original.

**. 2. Objetivos**

**2.1 Objetivo general**

Desarrollar un programa en C++ utilizando el framework Qt, capaz de **reconstruir una imagen original** que ha sido sometida a transformaciones bit a bit (XOR y rotación) y enmascaramientos, a partir de los datos proporcionados (imagen transformada, imagen aleatoria, máscara y archivos de enmascaramiento)

#### **2.2 Objetivos Específicos**

1. **Diseñar e implementar funciones para operaciones a nivel de bits**, tales como:

* XOR entre imágenes.
* Rotaciones (a izquierda y derecha) de bits en los valores RGB de cada píxel.
* Desplazamientos en arreglos de pixeles.

1. **Leer y procesar los archivos de enmascaramiento** (.txt) para:

* Extraer el desplazamiento S.
* Comparar las sumas resultantes con los valores obtenidos al aplicar la máscara a la imagen transformada.
* Verificar si una imagen intermedia coincide con una etapa específica del proceso de transformación.

1. **Diseñar un algoritmo para identificar el orden de las transformaciones aplicadas** (por ejemplo, XOR o rotación), de forma que permita:

* Deshacer dichas transformaciones en orden inverso.
* Obtener una reconstrucción precisa de la imagen original.

1. **Integrar el sistema completo en una aplicación funcional**, que:

* Reciba como entrada las imágenes ID, IM, la máscara M, y los archivos .txt de enmascaramiento.
* Aplique y verifique transformaciones.
* Restaure la imagen original IO y exporte el resultado como imagen BMP.

1. **Aplicar buenas prácticas de programación**, incluyendo:

* Uso de punteros y arreglos dinámicos para manipulación eficiente de imágenes.
* Manejo correcto de archivos binarios y de texto.
* Estructuración modular del código, separando claramente lectura, procesamiento y escritura.

1. **Elaborar documentación clara del desarrollo**, que incluya:

* Análisis del problema.
* Diseño de algoritmos y flujo de tareas.
* Justificación de decisiones técnicas.
* Problemas encontrados y soluciones adoptadas.

1. **Publicar el desarrollo en un repositorio público con control de versiones**

* Que muestre la evolución del proyecto a través de commits regulares.

1. **Realizar un video explicativo**, que muestre:

* El análisis y diseño de la solución.
* La demostración funcional de la aplicación.
* La explicación del código fuente y las decisiones de implementación.
* **3. Análisis de las transformaciones**

**3.1 Desplazamiento de bits**

Cuando se desplazan los bits de un byte hacia la derecha o izquierda, sabemos que el máximo rango de movimiento puede ser de 8 bits nos dimos de cuenta que los bits "empujados" se pierden. Esta operación **no es reversible** porque no existe información para recuperar lo perdido.

**Ejemplo de desplazamiento a la derecha (3 bits):**

Original (1 byte): 10111101 → Desplazado 3 bits a la derecha: 00010111 (Los ceros entran por la izquierda)

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Si se quiere invertir esos mismos 3 bits a la izquierda: 00010111 → 10111000 (**No se recuperó el original)** Porque el numero 001 no quedo guardado en ninguna parte.

Diagrama

Descripción generada automáticamente

**Conclusión:** Esta operación **pierde información**, por lo tanto, **no se usará** en la decodificación. si esta información se pudiera almacenar en algún lugar, deberían ser específicos en un archivo aparte para posteriormente incorporarlos a la imagen original, sin embargo, este no es el caso.

**3.2 Rotación de bits**

La rotación mueve los bits dentro del byte sin perder ninguno, Esta transformación se les aplica a todos los bytes RGB de la imagen. Por ejemplo, si se rota a la derecha 3 bits, los 3 últimos bits se mueven al principio.

**Ejemplo:**

Original (1 byte): 10111101

Rotando derecha (3 bits): 11101101→ Rotado izquierda (3bits): 10111101 **(Se recupera el original)**

**Conclusión:** Esta operación **es reversible** y segura para encriptación o desencriptación.

**3.3 XOR con imagen de ruido**

|  |  |  |
| --- | --- | --- |
| *A* | *B* | *A* XOR *B* |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

La operación XOR compara dos bytes (uno de la imagen original y otro de la imagen de ruido). Al aplicar XOR dos veces con el mismo patrón se recupera el original:

M/O: byte de la imagen original

M/D: byte de la imagen ruido

R1: M/O ^ M/D

R2: R1 ^ M/D = M/O

**Ejemplo: XOR con imagen de ruido**

Supongamos que estamos trabajando con **bytes** (8 bits), y tenemos lo siguiente:

* **M/O** (Imagen original): 11010110
* **M/D** (Máscara de ruido): 01101100

**Primer paso:** Aplicar XOR entre M/O y M/D

**Interfaz de usuario gráfica, Texto

Descripción generada automáticamente**

Este **R1** es lo que guardaríamos como imagen transformada o “encriptada”.

**Segundo paso:** Volver a aplicar XOR con la misma máscara (M/D)

**Texto

Descripción generada automáticamente**

La operación XOR tiene una propiedad muy útil que es: A ^ B ^ B = A

Es decir, si se aplica XOR dos veces con el mismo valor, el segundo XOR deshace el primero.

**Conclusión:** Es una operación **segura y reversible**. Se puede aplicar varias veces, siempre que se conozca el patrón (M/D).

**3.4 Conclusiones de las transformaciones**

Tras examinar detenidamente cada tipo de transformación, concluimos que únicamente dos de ellas permiten modificar la imagen sin pérdida de información: la rotación y la operación **XOR**. En cambio, el desplazamiento de bits elimina parte de la información original, lo que impide reconstruir la imagen tal como era. Por tanto, esta última puede considerarse una transformación engañosa, ya que imposibilita revertir el proceso.

* **Transformaciones válidas**: XOR y rotaciones.
* **Transformaciones descartadas**: desplazamientos de bits.

Este análisis nos dio una base sólida para iniciar la codificación del algoritmo de desencriptación, asegurándonos de enfocarnos exclusivamente en las operaciones reversibles. Así, se procedió con la implementación de funciones específicas para rotación visual (izquierda y derecha) y operación XOR, como parte fundamental del proceso.

* **4. Análisis del enmascaramiento**

La máscara es un fragmento de imagen que se suma (elemento a elemento) a una sección de la imagen transformada. Esa sección se determina con una **semilla aleatoria**.

La máscara puede aplicarse en cada etapa del proceso de transformación. Si el número de pasos es reducido y la máscara es de tamaño pequeño, su efecto sobre la imagen original será leve si no se elimina. Sin embargo, cuando se realizan múltiples transformaciones sucesivas, la máscara puede acumular alteraciones que afectan significativamente la calidad de la imagen al intentar recuperarla. En otras palabras, entre más se transforme sin eliminar la máscara, mayor será la distorsión acumulada.

**Enmascaramiento:**

S(k) = ID(k + s) + M(k)

Esto genera archivos .txt con resultados que se pueden comparar en cada paso de transformación.

**Importancia:** Si se quita la máscara correctamente (restando), se puede verificar si el resultado intermedio coincide con los archivos de pista.

Texto

Descripción generada automáticamente

\* datosMask[i] = S(k) → lo que estaba guardado en el archivo .txt

\* imgMasc[i] = M(k) → lo que está en la imagen de la máscara BMP

\* Al hacer S(k) - M(k) se obtiene:

ID(k + s) = datosMask[i] - imgMasc[i]

Esto revierte el enmascaramiento, lo cual permite verificar si la imagen transformada tiene coincidencia con ese resultado.

* **5. Estructura del algoritmo**

**Proceso de encriptación original (ejemplo):**

Paso 1: M\_O → TA → M → TXT1 → M\_D1

Paso 2: M\_D1 → TA → M → TXT2 → M\_D2

Paso 3: M\_D2 → TA → M\_D3 (final)

**Entradas disponibles:**

* Imagen final distorsionada (M\_D3)
* Archivos .txt (TXT1, TXT2)
* Imagen de ruido
* Máscaras M

Diagrama

Descripción generada automáticamente

**Meta:** Aplicar las transformaciones e ir comparando los resultados intermedios con los archivos TXT para descubrir el orden correcto. Además, este proceso implica realizar ingeniería inversa, ya que debemos partir del paso 3 y retroceder hasta llegar al paso 1.

* **6. Metodología y decisiones técnicas**

El desarrollo del Desafío I no fue un simple ejercicio de codificación. Representó una experiencia profunda, tanto a nivel técnico como personal, y exigió habilidades en análisis, lógica y programación con C++ utilizando el framework Qt. A continuación, se detallan las decisiones metodológicas más importantes tomadas durante el proyecto.

El desafío consistía en desencriptar una imagen BMP transformada mediante operaciones como XOR, rotaciones de bits y la aplicación de una máscara. Dado que las pistas estaban en varios archivos .txt, con una estructura poco convencional, el proyecto se convirtió en una especie de ejercicio de ingeniería inversa, en donde se debían probar múltiples combinaciones hasta dar con la transformación correcta.

Se adoptó una estrategia de desarrollo modular. El código fue construido por etapas, cada una manejada en funciones específicas para facilitar su comprensión y depuración. Una de las primeras decisiones técnicas fue utilizar memoria dinámica y punteros (unsigned char\*), ya que se debía operar byte por byte sin usar la librería STL, porque estaba restringido por las condiciones del reto.

Otra decisión clave fue el uso de QImage para la carga de las imágenes BMP. Este enfoque permitió convertir todas las imágenes al formato RGB888 (codificación del color, representada por 8 bits de información) y así trabajar directamente con sus valores RGB (rojo, verde y azul). Se descartó leer el archivo BMP desde cero porque implicaría manipular el encabezado y estructura binaria del archivo, lo cual no aportaba al objetivo del desafío y aumentaría innecesariamente la complejidad.

Las funciones que se implementaron fueron:

| **Función** | **Descripción** |
| --- | --- |
| xorImagen(...) | Aplica XOR entre dos arreglos de imágenes |
| rotarBitsDerecha(...) | Rota n bits hacia la derecha por byte (por canal RGB) |
| rotarBitsIzquierda(...) | Rota n bits hacia la izquierda (operación inversa) |
| loadPixels(...) | Carga imagen BMP como arreglo de bytes RGB |
| exportImage(...) | Guarda imagen modificada como archivo BMP |
| loadSeedMasking(...) | Carga semilla y datos RGB de un archivo .txt |
| comparar(...) | Hace la comparación byte a byte para validar |

La lógica del algoritmo consiste en aplicar una transformación y eliminar el efecto de la máscara mediante una resta. Dicha resta se realiza con los datos del archivo .txt, los cuales se leen y se almacenan en un arreglo dinámico. Cabe aclarar que los datos de la imagen de la máscara también se exportan y se guardan en otro arreglo dinámico.

A continuación, se lleva a cabo la resta entre los datos del .txt y los de la máscara, y el resultado de dicha operación se compara con los datos (bytes) de la imagen a la que se le aplicó a la transformación, comenzando desde la posición indicada por la semilla.

Si no hay coincidencia, se intentan nuevas transformaciones (rotaciones de 1 a 8 bits, hacia la izquierda o hacia la derecha), hasta encontrar una coincidencia. Si no se encuentra ninguna, se imprime un mensaje en la consola.

* **7. Resultados, dificultades y validación**

**Resultados esperados**

El algoritmo fue exitosamente probado con los archivos de los casos 1 y 2 del desafío. En todos los escenarios en los que se pudo determinar la transformación correcta, el programa desencriptó la imagen correctamente. Al aplicar XOR o una de las rotaciones, y al quitar la máscara (restando), el resultado coincidía con los valores de la imagen transformada. La imagen final era exportada en formato BMP y verificada visualmente, asegurando la fidelidad de los datos.

**Dificultades encontradas**

Durante el desarrollo se enfrentaron diversos desafíos técnicos:

* **Rutas con tildes o espacios:** Fallos al cargar archivos debido a nombres con caracteres especiales. Se solucionó renombrando carpetas y ajustando rutas con barras convencionales /.
* **Confusión entre tipos de datos:** Se mezclaban unsigned int (del archivo .txt) con unsigned char (de las imágenes). Esto generaba dudas o confusiones con las operaciones, pero se resolvieron con casting y validaciones (por ejemplo, añadiendo `if (resta < 0)` y `if (resta > 255) al hacer las restas).
* **Desbordamiento de índices:** En algunas pruebas, la máscara o los datos del archivo .txt excedían el tamaño del arreglo. Se resolvió validando los límites de los arreglos antes de operar sobre ellos.
* **Rotaciones que no coincidían:** Algunas funciones de rotación no daban los resultados esperados. Para corregir esto, se construyeron ejemplos manuales bit a bit y se comprobó que el desplazamiento fuera “visual”, es decir, circular, y no una simple operación lógica.
* **Comparaciones parciales:** Era importante que la comparación se hiciera desde la semilla y solo por la cantidad de bytes que tiene la máscara. Esto implicó implementar una función comparar(...) específica para ese propósito.

Texto

Descripción generada automáticamente

**Validación**

Los resultados se validaron tanto visualmente (viendo la imagen desencriptada) como numéricamente, comparando los valores RGB con los del archivo .txt. En todos los casos exitosos, se cumplió con la lógica del desenmascaramiento inverso:

* 1. imgOrig XOR imgGauss = resultado
  2. datosMask – imgMasc = esperado
  3. resultado = esperado → Transformación correcta

O en el caso de rotaciones:

* 1. imgOrig APLICAR (rotarBitsIzquierda O rotarBitsDerecha) = resultado
  2. datosMask – imgMasc = esperado
  3. resultado = esperado → Transformación correcta

Esto confirmaba que se había hallado la transformación correcta.

* **8. Conclusiones**

Este desafío fortaleció habilidades clave en programación con C++, como el uso de punteros, la rotación de bits, las operaciones a nivel de bytes, la manipulación de memoria dinámica y el manejo de estructuras de imagen. No se trató únicamente de codificar, sino también de razonar cada paso, validar hipótesis y evaluar distintas alternativas.

El enfoque modular del código y la capacidad de comparar datos parcialmente desde una posición específica (la semilla) fueron aspectos fundamentales para lograr un algoritmo exitoso. Además, se construyó un programa claro y estructurado, reutilizable y adaptable a tareas similares.

Este proceso permitió afianzar conocimientos técnicos y desarrollar una forma de pensar orientada a la resolución de problemas reales, combinando análisis lógico, pruebas constantes y una implementación cuidadosa.