**Informe Técnico – Recuperación de Imagen BMP Mediante Operaciones Bit a Bit**

**Introducción**

Este proyecto tiene como finalidad recuperar una imagen BMP original (I\_O.bmp) a partir de una imagen distorsionada (I\_D.bmp) utilizando operaciones bit a bit, máscaras, rotaciones y un sistema de verificación con archivos auxiliares (M0.txt, M1.txt, M2.txt). El código fue desarrollado en C++ haciendo uso de la biblioteca Qt para la manipulación de imágenes, y está organizado en módulos que permiten separar claramente la lógica de procesamiento, verificación y utilidades.

1. **Análisis del problema**
2. **Tareas definidas en el desarrollo de los algoritmos**
3. **Algoritmos implementados**
4. **Problemas afrontados en el desarrollo**
5. **Evolución de la solución y consideraciones para tener en cuenta**
6. Análisis

Para desarrollar este proyecto fue esencial comprender en profundidad las transformaciones aplicadas sobre la imagen original (I\_O.bmp) y cómo revertirlas de manera precisa. Estas transformaciones incluían operaciones bit a bit como la rotación de bits y la operación XOR, las cuales alteran directamente los valores de los píxeles en la imagen.

Además de invertir estas transformaciones, también fue necesario entender el proceso de enmascaramiento aplicado en conjunto con una imagen máscara (M.bmp). Este enmascaramiento implicaba sumar ciertos valores de la imagen transformada con los de la máscara, generando así archivos de rastreo (M0.txt, M1.txt, M2.txt) que contienen resultados numéricos específicos. Estos archivos jugaron un papel clave en la verificación de cada etapa del proceso de distorsión y recuperación.

En síntesis, el desarrollo del proyecto requirió no solo revertir transformaciones a nivel de bits, sino también interpretar y utilizar de forma efectiva la información contenida en los archivos .txt para validar la integridad de cada paso. Esta combinación de análisis binario y verificación numérica fue fundamental para garantizar la recuperación correcta de la imagen original.

1. Tareas definidas

Durante el desarrollo del proyecto, las tareas principales se organizaron en función de los objetivos técnicos requeridos para revertir el proceso de distorsión aplicado a una imagen BMP. A continuación, se detallan las actividades realizadas:

1. **Carga y análisis de imágenes**:  
   Se desarrollaron funciones para cargar imágenes BMP de 24 bits y extraer la información de píxeles necesaria para operar a nivel de bytes.
2. **Diseño e implementación de operaciones bit a bit**:  
   Fue necesario implementar manualmente operaciones básicas como desplazamientos y rotaciones de bits (izquierda y derecha), además de la operación XOR entre bytes, dado que estas eran las transformaciones aplicadas originalmente a la imagen.
3. **Reversión de la distorsión aplicada**:  
   Se estableció un orden inverso de ejecución para deshacer los efectos de la distorsión. Esto implicó aplicar las operaciones contrarias a las originalmente usadas, respetando el orden correcto.
4. **Verificación mediante archivos .txt**:  
   Se utilizó la información de los archivos M0.txt, M1.txt y M2.txt para validar si cada paso del proceso de recuperación coincidía con los resultados esperados. Para ello, fue necesario implementar un módulo verificador que comparara sumas parciales entre porciones específicas de la imagen transformada y la máscara.  
   Estas sumas se calculan mediante un enmascaramiento definido por una máscara M de dimensiones i × j (donde i ≤ m y j ≤ n) con tres canales (RGB).  
   Después de cada transformación a nivel de bits, se selecciona un desplazamiento aleatorio s dentro de la imagen transformada ID, y se calcula la suma para cada componente como:  
    **S(k) = ID(k + s) + M(k)**, para 0 ≤ k < i × j × 3.  
   El resultado de estas sumas verificaa si una imagen intermedia corresponde al paso correcto del proceso de recuperación.
5. **Modularización del código**:  
    El proyecto fue dividido en archivos .cpp y .h para separar correctamente la lógica principal de la verificación, las operaciones bit a bit y la funcionalidad de lanzamiento del programa (launcher.cpp).
6. **Exportación de resultados intermedios**:  
   Con el fin de facilitar pruebas y verificaciones visuales, se exportaron imágenes intermedias como P1.bmp, P2.bmp y la imagen final recuperada, permitiendo comprobar el progreso paso a paso.
7. Algoritmos

Los algoritmos desarrollados están centrados en transformaciones de bajo nivel sobre datos de imagen, utilizando operaciones bit a bit sobre valores de 8 bits (uint8\_t). A continuación, se describen los principales algoritmos:

1. **Operación XOR**  
   La función xor\_bits(a, b) permite realizar una XOR entre dos valores de 8 bits. Esta operación fue usada tanto para distorsionar como para recuperar la imagen original, aprovechando la propiedad de la XOR que permite revertir su efecto aplicándola dos veces con la misma clave.
2. **Rotaciones de bits**
   1. rotacion\_left(valor, n): Rota los bits de un byte hacia la izquierda en forma circular.
   2. rotacion\_right(valor, n): Rota los bits hacia la derecha en forma circular. Estas funciones permiten simular un cifrado sencillo sobre los datos de la imagen, alterando el orden interno de los bits.
3. **Desplazamientos simples**  
   Aunque no fueron parte central del algoritmo final, también se definieron shift\_left y shift\_right como herramientas complementarias para pruebas o futuras variantes del esquema de distorsión.
4. **Carga de archivos .txt con semilla**  
   El archivo loadSeedMasking() lee los archivos M0.txt, M1.txt y M2.txt, extrae la semilla de inicio, y los valores RGB resultantes de una suma entre una sección de la imagen y una máscara. Esta información se usa como traza para validar si una imagen intermedia es correcta.
5. **Comparación por suma de píxeles**  
   En el archivo verificador.cpp, se realiza una suma entre los valores RGB de una porción de imagen y una máscara para verificar coincidencias con los archivos .txt. Si las sumas coinciden, se confirma que ese paso fue ejecutado correctamente.
6. **Lanzador condicional (launcher.cpp)**  
   Se implementó un archivo de entrada que permite ejecutar el main de recuperación o el verificador, dependiendo de si la constante EJECUTAR\_VERIFICADOR está definida o no. Esto facilita pruebas modulares sin tener que recompilar constantemente.
7. Problemas afrontados

Durante el desarrollo del proyecto se presentaron varios desafíos tanto conceptuales como técnicos:

1. **Inversión de las transformaciones**  
   Uno de los principales retos fue entender cómo revertir las transformaciones aplicadas originalmente a la imagen. Si bien se conocía el orden de las operaciones (XOR, rotaciones, etc.), implementar sus inversas de forma correcta y en orden inverso tomó tiempo, ya que cada operación debía ser cuidadosamente adaptada para que su efecto fuera exactamente el opuesto.
2. **Modularización del código y archivos separados**  
   Otro desafío fue estructurar el proyecto en varios archivos .cpp y .h, especialmente para separar las funciones de manipulación de bits (bitops). Inicialmente, esto generó errores de compilación por problemas de vinculación entre los archivos. Fue necesario comprender bien el uso de cabeceras (.h) para declarar funciones y evitar duplicación o referencias cruzadas incorrectas.
3. **Uso de launcher.cpp**  
   Para facilitar la ejecución del proyecto, se investigó e implementó un archivo llamado launcher.cpp, cuyo propósito es centralizar la ejecución de distintos módulos del programa (por ejemplo, principal.cpp y verificador.cpp) mediante directivas de compilación. Al definir o comentar una línea (#define EJECUTAR\_VERIFICADOR), se puede cambiar fácilmente qué módulo se ejecuta sin modificar el main() de cada archivo. Esto resultó ser una herramienta útil para el control del flujo de ejecución durante el desarrollo y las pruebas.
4. **Implementación del enmascaramiento y verificación**  
   Finalmente, uno de los aspectos más complejos fue el desarrollo del algoritmo que verifica la correspondencia entre imágenes intermedias (P1, P2 y la imagen final recuperada) y los archivos de suma (M0.txt, M1.txt, M2.txt).  
    Implementar correctamente la lógica de enmascaramiento, incluyendo la suma por canal a partir de una posición aleatoria en la imagen (s), fue un reto tanto a nivel de comprensión como de programación. Requirió múltiples pruebas para garantizar que las sumas coincidieran exactamente con los valores registrados en los archivos .txt.
5. Evolución y consideraciones

A lo largo del desarrollo del proyecto, la solución evolucionó desde una idea general de inversión de transformaciones sobre una imagen BMP hasta una estructura modular, robusta y reutilizable. En un principio, las transformaciones se concebían como simples operaciones lineales, pero al profundizar en los detalles se hizo evidente la necesidad de modularizar cada parte del proceso, tanto para facilitar la depuración como para mejorar la escalabilidad del código.

Se pasó de una implementación monolítica a una organización basada en varios archivos fuente y cabeceras (bitops.cpp, verificador.cpp, principal.cpp, launcher.cpp), lo que permitió encapsular funcionalidades específicas como las operaciones a nivel de bit o el proceso de verificación.

Además, se incorporó un enfoque más sistemático para las pruebas: al utilizar un archivo launcher.cpp, se pudo controlar fácilmente qué parte del programa ejecutar sin alterar directamente los main() de los módulos principales, lo que mejoró significativamente el flujo de trabajo y la eficiencia en las pruebas.

En cuanto a las **consideraciones técnicas clave para una implementación correcta**, se destacan:

* La importancia de **respetar el orden inverso exacto de las transformaciones** aplicadas originalmente a la imagen para poder recuperar el contenido original de forma fiel.
* La necesidad de **entender y manejar correctamente las operaciones a nivel de bits**, especialmente las rotaciones circulares, que no se implementan igual que los desplazamientos comunes.
* La correcta implementación del **enmascaramiento con la máscara M** y el uso de una semilla s para calcular sumas parciales, respetando el rango de píxeles definido por las dimensiones de la máscara.
* La **gestión precisa de memoria dinámica** al cargar imágenes y máscaras, asegurando que se liberen los recursos correctamente para evitar fugas de memoria.

En resumen, la solución fue refinándose progresivamente hasta alcanzar un diseño organizado, funcional y capaz de cumplir con los objetivos propuestos. Este proceso permitió, además, adquirir una comprensión profunda del procesamiento de imágenes a nivel binario y del diseño modular en C++.