
FACULTAD DE MATEMÁTICAS

Trabajo fin de Grado

Grado en Matemáticas

# Librería sobre esteganografía en Haskell

Realizado por
Teresa Mariano Herzog

Dirigido por
Francisco Jesús Martín Mateos

Departamento
Ciencias de la Computación
e Inteligencia Artificial

Sevilla, Julio de 2023

## Abstract

Steganography is the art of hiding the fact that comunication is taking place. In this project we present an introduction to the concept of steganography, its history, and aplications, and a comparison with cryptography. Moreover, we give a brief description on different types of steganography and focus on the most popular one: Image Steganography. We will implement some algorithms of this type in Haskell and design an aplication that allows the user to hide or extract information in GIF images.

## Resumen

La esteganografía es el arte de esconder el hecho de que se está produciendo una comunicación. En este proyecto, presentamos una introducción al concepto de esteganografía, su historia y sus aplicaciones, y realizamos una comparación entre la esteganografía y la criptografía. Además, damos una breve descripción de los distintos tipos de esteganografía y nos centramos en el más popular: La esteganografía en imágenes. Implementaremos algunos algoritmos de este tipo en Haskell y diseñaremos una aplicación que permita al usuario esconder o extraer información en imágenes GIF.

# Índice general

# Índice de tablas

# Índice de figuras

# Índice de código

# CAPÍTULO 1

## Introducción

La palabra esteganografía [28] viene de las palabras griegas *"steganos"*, que significa "oculto" y *"graphein"*, que significa "escritura", es decir, significa "escritura oculta". La esteganografía es el arte y ciencia de escribir (y leer) mensajes ocultos de tal manera que solo el remitente y el destinatario sepan que el mensaje ha sido enviado. La esteganografía digital consiste en ocultar un documento, imagen, texto o vídeo en otro documento, mensaje o vídeo de manera que el objeto que lleva el mensaje sea indistinguible del original.

La formulación moderna está basada en el *"Problema del prisionero"* planteado por Simmons en 1983 [30]. En él, dos reclusos se quieren comunicar en secreto para planear su escape. El guardia de la cárcel les permite comunicarse siempre que pueda leer sus comunicaciones y cortará la comunicación si encuentra algo sospechoso. Debido a esto, es necesario que establezcan algún tipo de canal encubierto en sus mensajes y evitar ser engañados por mensajes modificados por el guardián.

Aunque la esteganografía y la criptografía comparten el mismo objetivo, difieren en la manera de usarlas. Ambas técnicas pretenden proteger información contra receptores indeseados, sin embargo, mientras que la criptografía protege el contenido del mensaje, la esteganografía esconde la existencia del mensaje.

## 1.1– Historia de la esteganografía

La idea y práctica de ocultar información se puede encontrar desde la Antigua China, en la que se escribían notas en pequeños trozos de seda y eran introducidos en pequeñas bolas cubiertas de cera que se tragaban los mensajeros. También podemos encontrar un ejemplo de esteganografía en *"Historias"*, del historiador griego Heródoto. En ella se cuenta cómo un noble, para comunicarse en secreto con su yerno, le tatuó el mensaje en la cabeza a un esclavo para ocultarlo con el pelo [28].

Más recientemente, durante la Segunda Guerra Mundial, los alemanes utilizaron la técnica del 'micropunto' [26], que consistía en reducir la información al tamaño de un punto. Sin embargo, no fue hasta 1985, con la aplicación de los ordenadores personales en problemas clásicos de la esteganografía, que se produjo el despegue de ésta [28].

## 1.2– Aplicaciones de la esteganografía [28]

El principal objetivo de la esteganografía es proteger la información. Por lo tanto, tiene numerosas aplicaciones en la seguridad de la información ya que permite ocultar el intercambio de mensajes.

Como la esteganografía permite esconder información en archivos, también se puede usar para sistemas de control de derechos de autor y marcas de agua. Algunos archivos pueden contener metadatos ocultos que refuerzan las herramientas de búsqueda de imágenes. También se utilizan en tarjetas de identidad inteligentes donde los datos de una persona se almacenan en su fotografía.

Además, se puede usar este concepto para la sincronización de audio y vídeo ocultando en éste la información del audio de manera que se puede tratar ambos como una sola entidad.

Algunas impresoras también hacen uso de la esteganografía para guardar en cada página datos como el modelo de la impresora o el tiempo y fecha de la impresión utilizando pequeños puntos amarillos que son casi invisibles para el ojo humano.

## 1.3– Tipos de esteganografía

La mayoría de los formatos digitales pueden ser utilizados en la esteganografía, sin embargo, son más útiles aquellos con un alto grado de redundancia. Los bits redundantes de un archivo son aquellos que se introducen con el fin de que no se pierdan datos ante posibles fallos. Debido a que estos bits son información repetida, el hecho de cambiarlos no produce alteraciones que sean detectadas fácilmente. Aunque otros formatos pueden ser usados para la esteganografía, los más propicios son las imágenes y los audios, que suelen tener un alto grado de redundancia. A continuación, explicaremos los formatos principales que se pueden usar en esteganografía expuestos en la siguiente imagen [26]:

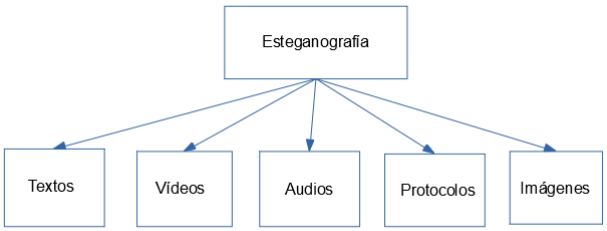


Figura 1.1: Tipos de esteganografía según el formato utilizado

### 1.3.1. Esteganografía en textos

La esteganografía en textos es considerada la más complicada debido a que el grado de redundancia es bajo, por lo que se puede ocultar una cantidad de información limitada. Además, incluso pequeñas modificaciones son notables. Sin embargo, este formato resulta interesante debido al amplio uso de los documentos en muchas organizaciones.

Dependiendo de la técnica para ocultar la información, la esteganografía en textos puede dividirse en tres categorías. A continuación explicaremos y daremos algunos ejemplos de cada categoría.

### Inserción a nivel de carácter [3]

La inserción a nivel de carácter oculta la información como caracteres del texto. Algunos métodos necesitan un documento tapadera donde se oculta el mensaje como caracteres marcando éstos en el documento, por lo que es imprescindible que todos los caracteres del mensaje estén en el texto. Un ejemplo es el marcado de caracteres, que consiste en resaltar los caracteres que forman parte del mensaje secreto cambiando la fuente, poniendo en negrita o en cursiva dichos caracteres. Otro ejemplo es cometer erratas a propósito o colocar algunos caracteres algo más arriba o más abajo. Como estos últimos errores son habituales, en principio no despierta sospechas, sin embargo si puede ser motivo de duda el uso excesivo de dichos "errores", por lo que es preferible usarlo para mensajes breves.

Otros métodos directamente crean un documento para esconder el mensaje, en vez de basarse en un documento tapadera ya creado. Por ejemplo, se puede escribir un documento tal que la letra en una posición concreta de cada palabra forma el mensaje oculto. Este método necesita de una persona experimentada ya que resulta difícil producir dicho documento. Otra técnica genera una lista de palabras de entre 6 y 15 caracteres. Las palabras son generadas respecto al valor decimal del carácter del mensaje secreto. Después se sustituyen algunos caracteres por signos de interrogación de manera que la longitud de cada palabra junto con la posición y el número de interrogaciones representan el carácter escondido. En vez de usar los signos de interrogación también se puede usar otra técnica en la que la longitud y la primera letra de cada palabra se utiliza para representarlo.

### Inserción a nivel de bit [3]

En los métodos de esta categoría se usa el código binario del mensaje que queremos ocultar. Por ejemplo, se pueden usar los sinónimos para cifrar la comunicación. Al sustituir una palabra, si utilizamos la primera palabra representa "0", mientras que el uso de la segunda representa "1". Para esta técnica, es necesario que tanto el emisor como el receptor tengan la lista completa de palabras con sus sinónimos para codificar y descodificar.

| **Palabra** | **Sinónimos** |
|---|---|
| homologar | equiparar, verificar |
| suceso | hecho, acontecimiento |
| dejar | partir, marchar |

Tabla 1.1: Ejemplo de palabras con sus respectivos sinónimos

Otra técnica de esta categoría es el desplazamiento de líneas. Dividimos el texto en grupo de tres líneas y usaremos la del medio de cada grupo para codificar. Si la línea está desplazada hacia arriba, codificamos un "0" y si lo está hacia abajo, codificamos un "1". El problema de este método es que solo podemos codificar un carácter cada tres líneas, por lo que debemos tener un texto más largo para codificar mensajes más largos.

Similarmente, se pueden añadir espacios en blanco extra entre palabras, frases o párrafos para esconder un mensaje. El principal problema es que hay algunos procesadores de texto que eliminan el espacio extra en el documento automáticamente.

#### Inserción de tipo mixto [3]

Esta categoría es una mezcla de las anteriores. Como en la inserción a nivel de bit, convierte el mensaje que queremos ocultar a su equivalente en código binario. Después, lo transforma en otros caracteres usando alguna función para luego insertar dichos caracteres en el texto, como en la inserción a nivel de carácter.

A esta categoría pertenece el algoritmo *"Generating Summary"* [24]. Para aplicar esta técnica, primero se clasifican las letras del abecedario en cuatro grupos: Las que son simétricas respecto al eje horizontal y vertical, las que lo son respecto al eje horizontal o al vertical y las que no son simétricas respecto a ninguno de los dos ejes. A cada grupo se le asocia unos bits, al primer grupo se le asocia el "11", al segundo el "01", al tercero el "10" y al cuarto el "00". Ahora, convertimos el mensaje que queremos ocultar a binario. A continuación, escogemos un texto cualquiera. Para esconder el mensaje, seleccionamos las frases cuyas letras empiezan por una letra del grupo que corresponde al bit que queremos escribir. Por ejemplo, si queremos esconder el bit "11", podemos seleccionar la frase "Hace un día soleado", que empieza por "H" que es una letra del primer grupo. Para recuperar el mensaje, se coge la primera letra de cada frase o, en el caso de que la primera palabra sea un artículo, la primera letra de la palabra a continuación. Se observa que bits corresponden a cada letra de acuerdo a la clasificación de antes y se convierte el texto de binario a su forma alfanumérica. A continuación, presentamos una tabla con la clasificación de las letras utilizada para esta técnica.

| Número de grupo | Nombre del grupo | Letras en el grupo | Bits |
|---|---|---|---|
| 1 | Letras simétricas respecto al eje horizontal y al eje vertical | H, I, O, X | 11 |
| 2 | Letras simétricas respecto al eje horizontal | B, D, E, K, S | 01 |
| 3 | Letras simétricas respecto al eje vertical | A, M, T, U, V, W, Y | 10 |
| 4 | Letras que no son simétricas respecto al eje vertical ni respecto al eje horizontal | C, F, G, J, L, N, P, Q, R, Z | 00 |

Tabla 1.2: Grupos basados en la simetría de las letras

### 1.3.2. Esteganografía en vídeos

Las técnicas de esteganografía en vídeos se pueden clasificar en tres grupos, las técnicas de dominio espacial, las de dominio de transformación y las técnicas basadas en el formato.

#### Técnicas de dominio espacial

En estas técnicas, la ocultación de datos se basa directamente en el valor del píxel. Un ejemplo que pertenece a esta categoría es *Frame Selected Approach* [25].

La técnica *Frame Selected Approach* usa vídeos digitales y la información se oculta en el fotograma seleccionado. Se trata de un método BPCS (Bit-Plane Complexity Segmentation) [29]. La esteganografía BPCS utiliza la descomposición en planos de bits[1] y las características de la vista en los seres humanos, y consiste en sustituir algunas regiones de los planos de bits con la información que se quiera ocultar, de manera que la calidad de la imagen no se ve afectada. Su principal ventaja es que proporciona gran seguridad, sin embargo, presenta el problema de que con este método se inserta poca cantidad de información.

#### Técnicas de dominio de transformación [25]

En este tipo de técnicas, se insertan los datos ocultos en la imagen transformada, que es obtenida a partir de métodos de transformación. Los métodos pertenecientes a esta categoría son la técnica basada en la transformada de coseno discreta[2] (DCT) y la técnica basada en la transformada de wavelet discreta[3] (DWT).

En las técnicas basadas en DCT, la imagen se divide en baja, media y alta frecuencia. Un método de este tipo son los códigos BCH correctores de errores. Los códigos BCH son una clase de códigos cíclicos que se construyen utilizando polinomios sobre un campo finito y en este método se encargan de realizar la codificación. El mensaje que queremos ocultar es insertado en los coeficientes DCT del fotograma. De esta manera, sólo se puede esconder una cantidad limitada de información, sin embargo, tiene una gran capacidad para evitar intrusiones y ocultar el mensaje.

La principal ventaja de la técnica DWT [21] es que la información que proporcionan las wavelets a veces permite reconstruir el todo a través de una suma simple de las componentes, lo cual es denominado como análisis multiresolución. Una técnica basada en la transformada de wavelet discreta es el Algoritmo de búsqueda Kanade Lucas Tomasi. En este método, el mensaje oculto se introduce en la frecuencia media y alta del coeficiente DWT. Se pueden encontrar más detalles en [27].

#### Técnicas basadas en el formato [25]

Estas técnicas se usan para realizar operaciones de formatos de vídeo específicos. Algunos formatos de vídeo que podemos encontrar son H.264/AVC, MPEG y FLV. Un ejemplo de una técnica basada en el formato es el Algoritmo de detección de cambio de escena. Este algoritmo usa la estimación del movimiento en MPEG y las operaciones se realizan en el dominio comprimido, es decir, en los coeficientes resultantes de la compresión de un vídeo. Este algoritmo tiene un bajo coste computacional pero da lugar a una calidad pobre del vídeo.

### 1.3.3. Esteganografía en audio

Actualmente, es común encontrarse con archivos de audio, lo que hace que este formato sea interesante para utilizar como medio para ocultar mensajes ya que no levantaría sospe-

[1] Un plano de bits es un conjunto de bits que se encuentran en una posición determinada en la codificación binaria de una imagen o un audio

[2] Transformada de coseno discreta: `https://es.wikipedia.org/wiki/Transformada_de_coseno_discreta`

[3] Transformada de wavelet discreta: `https://en.wikipedia.org/wiki/Discrete_wavelet_transform`

chas. Además, la mayoría de los esfuerzos en estegoanálisis [4] se centran en la esteganografía en imágenes, por lo que la esteganografía en audio es un terreno poco explorado.

En la esteganografía con este medio, se puede aprovechar los sonidos más altos para esconder sonidos más bajos o utilizar las distorsiones naturales del ambiente, que pasan desapercibidas para el oyente. A continuación, hablaremos de las tres categorías en las que se pueden clasificar algunos métodos de la esteganografía en audio: ocultación en dominio temporal, en dominio de transformación y en dominio codificado.

### Dominio temporal [4]

*Low-bit encoding* es una de las primeras técnicas usadas en esteganografía y corresponde a esta categoría. Consiste en esconder cada bit del mensaje en el bit menos significativo dentro del fichero de audio. Tiene la ventaja de que permite esconder una cantidad considerable de información y es fácil de combinar con otras técnicas para ocultar información. Sin embargo, esta técnica es vulnerable a ataques simples, como la adición de ruido y la compresión con pérdida del archivo de audio, que pueden llegar a destruir los datos.

*Echo hiding* es otro método que pertenece a esta categoría. Consiste en ocultar la información en señales de audio introduciendo un breve eco. Una vez añadido el eco, la señal resultante conserva las mismas características estadísticas y de percepción. Para ocultar los datos, se manipulan tres parámetros de la señal eco: la amplitud inicial, el retraso y la tasa de decaimiento[5]. El efecto de un retraso de un milisegundo entre la señal original y el eco es indistinguible. Además, se puede colocar la frecuencia del decaimiento y la amplitud por debajo de lo que puede percibir el oído humano de manera que se puede esconder información sin ser percibido.

### Dominio de transformación [4]

El sistema auditivo humano tiene ciertas flaquezas que pueden ser aprovechadas para la esteganografía. Una de las maneras es mediante el conocido como "efecto enmascarador", que consiste en que, al escuchar dos sonidos a la vez, el más intenso enmascara el sonido menos intenso aunque éste se encuentre en una frecuencia por encima del umbral de audición. Para conseguir que aquello que queremos ocultar en el audio sea inaudible se proponen varias técnicas que utilizan el efecto enmascarador directamente, modificando las regiones enmascaradas, o indirectamente, alterando levemente la señal de audio.

Un ejemplo de una técnica de este tipo es el Espectro de propagación (*Spread spectrum*). En este método se dispersa la información a través del espectro de frecuencias. El concepto espectro de propagación fue desarrollado para asegurar la recuperación de datos que puedan ser perdidos por un canal ruidoso. Esta recuperación es posible debido a que se producen copias redundantes de la señal. De esta manera, si el ruido corrompe el mensaje oculto, existen copias que permiten recuperar el mensaje completo.

[4] El estegoanálisis es la disciplina que se encarga del estudio de la detección de mensajes ocultos mediante esteganografía.

[5] Tasa de decaimiento: la velocidad con la que dicho eco va desapareciendo

### Dominio codificado [4]

Los codificadores de voz (como AMR, ACELP y SILK) y sus respectivas tasas de codificación[6] se utilizan a la hora de considerar el esconder información en la comunicación en tiempo real. La señal transmitida es codificada cuando pasa por los codificadores de acuerdo con su tasa de codificación y descomprimida cuando llega al receptor. Esto da lugar a que estas técnicas sean muy complicadas debido a que el hecho de que la señal no es la misma cuando se envía que cuando se recibe afecta a la precisión de la recuperación de la información oculta.

La principal ventaja de este tipo de técnicas es la seguridad que proporciona a los datos ocultos. Estos datos sobreviven la incorporación de ruidos. Sin embargo, la integridad de la información oculta se puede ver comprometida si existe un codificador/decodificador de voz en la red.

## 1.3.4. Esteganografía de protocolos [23]

Cuando hablamos de la esteganografía de protocolos, nos referimos a las técnicas que se utilizan para ocultar comunicaciones en mensajes y en protocolos de control usados en la transmisión de red. Este tipo de esteganografía se puede clasificar según las funciones de protocolo asociadas con las capas de OSI RM (Fig. 1.2).

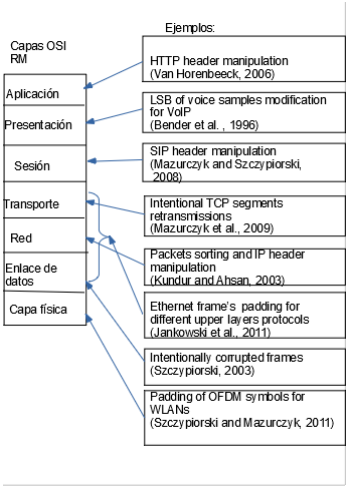


Figura 1.2: Tipos de esteganografía de protocolos según las capas de OSI RM

Sin embargo, nos vamos a centrar en la clasificación según el tipo de modificación de la unidad de datos de protocolo (PDUs):

### Modificación del protocolo PDU

Esta categoría se puede dividir en tres subcategorías: Modificación de la información de control de protocolo (PCI), de la unidad de datos de servicio (SDUs) y el mixto.

La información de control de protocolo es la comunicación entre entidades para coordinar una operación conjunta en un sistema. Los métodos basados en la modificación de

[6]La tasa de codificación se refiere a la cantidad de información que sale del codificador por segundo en el proceso de codificación

PCI tiene las ventajas de que tienen un gran ancho de banda esteganográfica y una implementación sencilla. Además, estos métodos no requieren de una sincronización entre el remitente y el receptor. Sin embargo, estos métodos son fáciles de detectar y puede deteriorar la funcionalidad del protocolo.

La unidad de datos de servicio es aquella que es derivada a la capa inferior en la pila de protocolos para posteriormente convertirse en la unidad de datos de protocolo de la misma. Los métodos basados en la modificación de los SDUs, al igual que los métodos basados en la modificación de PCI, no requiere de una sincronización entre el remitente y el receptor. Además, no son tan fáciles de detectar como los métodos basados en la modificación de los PCI. No obstante, tiene menor ancho de banda que los métodos basados en el PCI y son más difíciles de implementar. Además, con estos métodos existe la posibilidad de que haya un deterioro de la calidad del texto del mensaje.

Los métodos mixtos no necesitan que haya sincronización entre el emisor y el receptor y, además, son más difíciles de detectar. Al igual que los métodos basados en la modificación de la información de protocolo, disponen de un gran ancho de banda esteganográfica. Estos métodos son más difíciles de implementar que los anteriores y presenta el problema de que es posible que aumente el índice de errores de trasmisión.

#### Métodos que se basan en la modificación del tiempo y relaciones entre PDUs

Por ejemplo, estas modificaciones se pueden realizar reordenando paquetes de datos e introduciendo un retraso intencionado de los paquetes seleccionados. Estos métodos son fáciles de implementar y resultan complicados de detectar, sin embargo, estas técnicas no son ideales pues tienen poco ancho de banda esteganográfica y necesita que haya una sincronización entre el emisor y el receptor. Además, provoca que aumenten los retrasos en las trasmisiones.

#### Métodos híbridos

Estos métodos son una combinación de las dos categorías anteriores. Tienen numerosas ventajas ya que combina las ventajas de los métodos anteriores que son difíciles de detectar, no requieren de sincronización entre el emisor y el receptor, tienen un gran ancho de banda esteganográfica y son sencillos de implementar. Sin embargo, no evita la posibilidad de que den lugar a un deterioro de la calidad del texto del mensaje.

### 1.3.5. Esteganografía en imágenes

En nuestro día a día podemos encontrar en internet numerosas imágenes, ya sea a través de búsquedas o que compartamos a través de redes sociales. Además, las imágenes resultan un medio óptimo para la esteganografía por el alto nivel de redundancia de éstas. Debido a estos factores, la esteganografía en imágenes es la más común y la que más se ha estudiado. Las imágenes vienen en diferentes formatos y algunos se utilizan para aplicaciones determinadas. Para estos distintos formatos, existen algoritmos diseñados para ellos. Los distintos métodos pueden clasificarse en función de si se realizan los cambios en el dominio espacial o en el dominio de frecuencia de una imagen. A continuación, comentaremos brevemente ambos tipos:

### Dominio espacial [2]

Los métodos de este tipo se basan en la localización de los píxeles de una imagen. Insertan los bits del mensaje en el bit menos significativo de cada píxel para evitar que se aprecien los cambios en la imagen. Resultan ser más sencillos de implementar y pueden esconder mayores cantidades de información, pero también son más fáciles de detectar. Estas técnicas son más adecuadas para ser aplicadas en imágenes con formato como GIF o PNG y el método empleado depende de dicho formato. El algoritmo más común perteneciente a esta categoría es el LSB-replacement.

### Dominio de frecuencia [2]

Estos métodos esconden el mensaje en áreas significativas de la imagen y resultan menos vulnerables que los de dominio espacial. Utiliza varias técnicas de transformación para modificar la imagen donde queremos esconder el mensaje. Uno de los ejemplos más significativos es el algoritmo JSteg, usado para imágenes en formato JPEG.

### Híbridos [2]

Estos algoritmos pueden combinar los dominios de frecuencia y el espacial, o combinar alguno de ellos con otras técnicas como algoritmos de optimización o de criptografía, para llegar a mejores resultados de los que se podría conseguir si se aplicasen solos.

### Dominio espacial o de frecuencia [26]

También hay técnicas que, dependiendo de su implementación, pueden clasificarse como métodos en el dominio espacial o en el dominio de frecuencia.

# CAPÍTULO 2

## Esteganografía en imágenes

Este capítulo se centrará en definir algunos conceptos y algoritmos relacionados con la esteganografía en imágenes. Este tipo de esteganografía, como se ha mencionado anteriormente, es el más popular. La imagen en la que se va a esconder un mensaje, antes de que se realice esta acción, se denomina *cover image*. Por otro lado, cuando se ha efectuado la inserción del mensaje en la imagen mediante algún algoritmo esteganográfico, a la imagen resultante se le denomina como *stego image*. Para que la esteganografía cumpla su objetivo, estas dos imágenes deben ser idénticas para el ojo humano, de tal manera que no se pueda apreciar que se haya hecho ninguna modificación a la *cover image*. A continuación, se incluye un esquema con el modelo general de la esteganografía en imágenes:

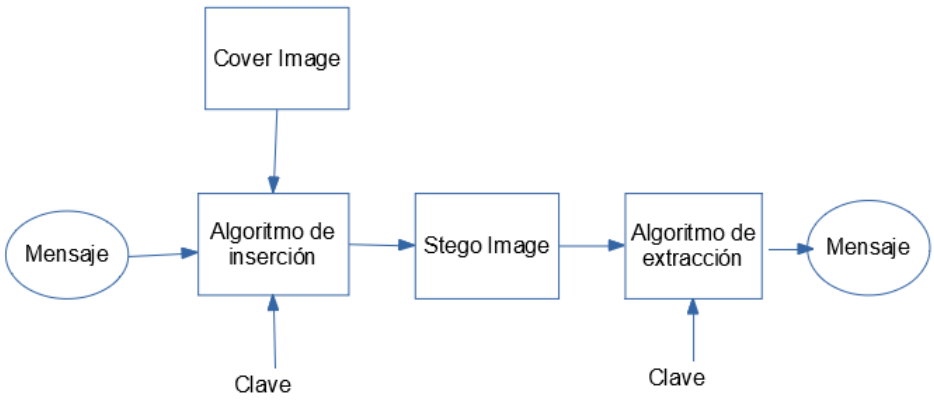


Figura 2.1: Esquema general de la esteganografía en imágenes

## 2.1– Conceptos relacionados con las imágenes a tener en cuenta

Para un ordenador, una imagen es un conjunto de números que representan las intensidades de la luz en las distintas partes de la imagen. Esta representación numérica forma

una cuadrícula y los puntos individuales se llaman píxeles. Estos píxeles se muestran horizontalmente fila por fila.

La profundidad de bits o de color es el número de bits en cada píxel. El menor número de profundidad de color es 8, es decir, se usan 8 bits para describir el color de cada píxel. Las imágenes monocromáticas y en escala de grises tienen profundidad de color 8. Las imágenes digitales de color se guardan en archivos de 24 bits y usan el modelo RGB. Los colores rojo, verde y azul, son representados por 8 bits cada uno. Cuanto mayor sea la profundidad de bits, mayor será el tamaño del archivo. El octavo bit es conocido como el bit menos significativo.

La compresión de una imagen tiene un papel fundamental a la hora de elegir qué algoritmo esteganográfico se debe utilizar. Para facilitar la transmisión de imágenes con una gran profundidad de bit a través de internet es necesario reducir su tamaño. Para ello, se utiliza la compresión, que son una serie de técnicas que utiliza fórmulas matemáticas con este objetivo. Existen dos tipos de compresión: con pérdida y sin pérdida. La compresión con pérdida es utilizada por ejemplo en imágenes con formato JPEG. Este tipo de compresión descarta información de la imagen original que no es necesaria, es decir, aquella información que, si se desecha, no daría lugar a cambios perceptibles por el ojo humano. En el caso de la compresión sin pérdida, no se descarta ninguna información, sino que se representan los datos con fórmulas matemáticas.

## 2.2– Algoritmos para la esteganografía en imágenes

Es necesario notar que, a la hora esconder un mensaje con estos algoritmos, se tiene en cuenta la codificación binaria de éste, por lo que se hablará de los "bits del mensaje".

### 2.2.1. LSB-replacement [22] [26]

La inserción de la información de este algoritmo se realiza a nivel de la codificación de colores, sustituyendo el bit menos significativo (el último) por un bit del mensaje. Como se ha visto anteriormente, cada uno de los colores de cada píxel de una imagen RGB se guardan en 8 bits. Por lo tanto, en cada píxel se puede guardar 3 bits del mensaje. Por ejemplo, si se tienen estos tres píxeles:

(1 1 1 1 1 1 0 1  1 1 1 1 1 1 1 0  1 1 1 1 1 1 0 0)
(1 1 1 1 1 1 0 1  1 1 1 1 1 1 1 0  1 1 1 1 1 1 0 0)
(1 1 1 1 0 1 1 1  1 1 1 1 1 0 0 0  1 1 1 1 0 1 0 1)

Al esconder el mensaje `11011100`, la codificación de los colores de estos píxeles quedarían de la siguiente manera, donde los bits en negrita corresponden con los bits del mensaje:

(1 1 1 1 1 1 0 **1**  1 1 1 1 1 1 1 **1**  1 1 1 1 1 1 0 **0**)
(1 1 1 1 1 1 0 **1**  1 1 1 1 1 1 1 **1**  1 1 1 1 1 1 0 **1**)
(1 1 1 1 0 1 1 **0**  1 1 1 1 1 0 0 **0**  1 1 1 1 0 1 0 1)

Estos cambios son imperceptibles para el ojo humano, ya que, al cambiar este bit, la intensidad del color varía, como mucho, un nivel. Este algoritmo, aunque es fácil de implementar, también resulta sencillo de detectar. Esto se debe a que la incrustación es asimétrica, ya que no existe la misma probabilidad de incrementar un valor que de decrementarlo. A la hora de cambiar el bit menos significativo, tenemos que si el valor es

par (0) y queremos cambiarlo por un 1, el efecto que produce es el de añadir 1 al valor del número binario, y si el valor es impar (1) y se cambia por un 0, el efecto es el de restar 1. Esta asimetría provoca que se produzcan anomalías en la distribución estadística de los bits. Una manera de dificultar la obtención del mensaje oculto es añadir una clave secreta que indique en qué bits se encuentra escondido el mensaje.

En su forma más simple, este algoritmo se utiliza en imágenes con formato BMP, que es de compresión sin pérdida. Sin embargo, estas imágenes son poco habituales en internet, por lo que pueden provocar sospecha. Es por esto que se ha adaptado este algoritmo para otros formatos, como el formato GIF. Las imágenes GIF son imágenes indexadas donde los colores se guardan en una paleta. Cada píxel se representa por un byte y el dato de un píxel es un índice de la paleta. Esta característica de las imágenes GIF hace que con variaciones mínimas de color se puedan producir cambios significativos en el tamaño de la paleta. Una posible solución para evitar esto es utilizar imágenes en escala de grises, ya que la profundidad de bits es 8 y los cambios de color son más graduales, de manera que son más difíciles de detectar.

### 2.2.2. LSB-matching [22]

Como se ha visto, el algoritmo anterior introduce anomalías estadísticas. Para evitar esto, se utiliza el algoritmo LSB-matching, que ofrece los mismos resultados que el algoritmo LSB-replacement. El cambio principal con respecto a este algoritmo es que en el algoritmo LSB-matching no se sustituye directamente el bit menos significativo por un bit del mensaje, sino que, si hay que cambiar este bit, se suma 1 ó -1 aleatoriamente a la codificación binaria del color. Volviendo al ejemplo anterior, se tiene la siguiente codificación:

(1 1 1 1 1 1 0 1  1 1 1 1 1 1 1 0  1 1 1 1 1 1 0 0)
(1 1 1 1 1 1 0 1  1 1 1 1 1 1 1 0  1 1 1 1 1 1 0 0)
(1 1 1 1 0 1 1 1  1 1 1 1 1 0 0 0  1 1 1 1 0 1 0 1)

Ahora, para esconder el mensaje `11011100`, hay que cambiar 4 bits, por lo que, se generan aleatoriamente tantos valores 1 ó -1 como bits haya que cambiar. Por ejemplo, se considera la siguiente lista: `[1,-1,-1,-1]`. Por lo tanto, las modificaciones son las siguientes:

(1 1 1 1 1 1 0 1  1 1 1 1 1 1 1 0 **(+1)**  1 1 1 1 1 1 0 0  )
(1 1 1 1 1 1 0 1  1 1 1 1 1 1 1 0  **(-1)**  1 1 1 1 1 1 0 0 **(-1)**)
(1 1 1 1 0 1 1 1 **(-1)**  1 1 1 1 1 0 0 0  1 1 1 1 0 1 0 1  )

Queda la siguiente codificación, donde los números en negrita corresponden al mensaje:

(1 1 1 1 1 1 0 **1**  1 1 1 1 1 1 1 **1**  1 1 1 1 1 1 0 **0**)
(1 1 1 1 1 1 0 **1**  1 1 1 1 1 1 1 0 **1**  1 1 1 1 1 0 1 **1**)
(1 1 1 1 0 1 1 **0**  1 1 1 1 1 0 0 0 **0**  1 1 1 1 0 1 0 **1**)

### 2.2.3. JSteg [22] [26]

#### Compresión de los archivos JPEG

Para comprimir un archivo JPEG, primero se transforma la representación de color RGB en la representación YUV, donde Y corresponde a la luminosidad. y U y V a los

colores. El ojo humano detecta mejor los cambios en la luminosidad que en los colores, por lo que en la compresión puede reducirse la información cromática a la mitad.

A continuación, se transforma la imagen usando la Transformada de Coseno Discreta (DCT). Los píxeles se agrupan en bloques de 8x8 y, después, se transforman en 64 coeficientes DCT por cada bloque. A continuación, se dividen estos coeficientes por unas matrices definidas conocidas como matrices de cuantificación[1], redondeando al número entero más cercano. Estas matrices reducen la información en los componentes de alta frecuencia, que son las que peor distingue el ojo humano. Los coeficientes resultantes son cifrados con la codificación Huffman[2], que es un algoritmo de compresión sin pérdida.

#### Esteganografía en los archivos JPEG

Debido a la compresión aplicada a los archivos JPEG, se pensaba que la esteganografía en las imágenes JPEG no sería posible, ya que los bits redundantes son eliminados y se temía que se eliminara el mensaje. Sin embargo, se ha conseguido explotar las propiedades de la compresión JPEG para poder desarrollar un algoritmo esteganográfico en este tipo de archivos.

La compresión de los archivos JPEG se divide en fases con pérdida y fases sin pérdida. La transformación mediante la Transformada de Coseno Discreta y la cuantificación descritas en el apartado anterior forman parte de las fases con pérdida, y la codificación de Huffman, a la fase sin pérdida. Por lo tanto, se puede esconder un mensaje entre estas dos partes de la compresión. En el caso del algoritmo Jsteg, se reemplaza el bit menos significativo por un bit del mensaje después de la cuantificación.

Es necesario destacar que no es conveniente guardar el mensaje en los coeficientes nulos. La manera en la que JPEG almacena los datos evita guardar los ceros, por lo que si se oculta información en estos coeficientes, tendrán que ser almacenados y el tamaño del archivo crecería.

### 2.2.4. Matrix Embedding [5] [32]

Este algoritmo se basa en la premisa de que cuanta más información se introduzca en una imagen, más fácil de detectar serán estas modificaciones. En este algoritmo, se pretende almacenar la mayor cantidad de información posible, modificando el menor de número de bits.

Para esconder n bits utilizando este algoritmo, primero, se escogen los $2^n - 1$ bits menos significativos. Al vector columna cuyos elementos son estos bits, le llamaremos $v$. Por otro lado, se tiene una matriz $M$, de dimensión $n \times 2^n - 1$, cuyas columnas están formadas por vectores de dimensión n no nulos. Esta matriz surge de exigir que el número de cambios en los bits menos significativos sea mínimo. La idea es que se debe cumplir que $Mv = m$, donde $m$ es el vector con los bits del mensaje. En caso de que no se cumpla, se detecta qué bit de la imagen hay que cambiar para que se cumplan las ecuaciones.

Veamos un ejemplo para un bloque de dos bits perteneciente al mensaje que se quiere ocultar. Sean $m_1$ y $m_2$ los bits de dicho bloque. Se consideran los 3 bits menos significativos que se van a utilizar para esconder el bloque, a los que se les va a denominar $o_1$, $o_2$ y $o_3$. Entonces, se tiene que se debe cumplir que:

[1] Para más información, consultar `https://hmong.es/wiki/Quantization_(image_processing)`
[2] Codificación Huffman: `https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/`

$$\begin{bmatrix} m_1 \\ m_2 \end{bmatrix} = \begin{bmatrix} o_1 \oplus o_3 \\ o_2 \oplus o_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} o_1 \\ o_2 \\ o_3 \end{bmatrix}$$

Donde el símbolo "$\oplus$" hace referencia a la suma binaria. En caso de que se cumpla esta ecuación, no se realiza ningún cambio. En cambio, si no se cumple que $m_1 = o_1 \oplus o_3$ ó $m_2 = o_2 \oplus o_3$, se debe cambiar el bit $o_1$ ó $o_2$ respectivamente. Si no se cumple ninguna de las dos ecuaciones, se cambia el bit $o_3$.

## 2.3– Comparación de los algoritmos

Para comparar los distintos algoritmos, se van a tener en cuenta los siguientes criterios:

- **Invisibilidad**: La invisibilidad del algoritmo esteganográfico es la cualidad más importante de éste, ya que el objetivo de la esteganografía es que, a simple vista, no se detecte que se está dando una comunicación entre dos partes.
- **Capacidad de guardar información**: Como el objetivo de la esteganografía es la comunicación oculta, es importante que los algoritmos esteganográficos permitan ocultar gran cantidad de información.
- **Resistencia ante ataques estadísticos**: El análisis estadístico tiene como objetivo detectar la información oculta a través de estudios estadísticos aplicados a los datos de una imagen. Algunos algoritmos dejan una "huella" que puede ser detectada mediante el análisis estadístico. Lo ideal sería que un algoritmo esteganográfico no dejara constancia de que ha sido usado.
- **Archivos no sospechosos**: Para no llamar la atención sobre la imagen stego, lo conveniente es que el formato del archivo sea común y no levante sospechas.

Veamos ahora estas cualidades en los algoritmos esteganográficos descritos en la sección 2.2:

- LSB-replacement en imágenes GIF:
  - Invisibilidad: Dependiendo de la imagen stego, la invisibilidad puede variar. Como se ha mencionado antes, es mejor utilizar imágenes en escala de grises
  - Capacidad de guardar información: Alta
  - Resistencia ante ataques estadísticos: Baja
  - Archivos no sospechosos: Las imágenes GIF no son las imágenes más frecuentes en internet. Un uso habitual de este formato puede levantar sospechas.
- LSB-matching en imágenes GIF:
  - Invisibilidad: Igual que en LSB-replacement
  - Capacidad de guardar información: Alta
  - Resistencia ante ataques estadísticos: Es menos susceptible a estos ataques que el algoritmo LSB-relacement
  - Archivos no sospechosos: Igual que en LSB-replacement en imágenes GIF.

- Jsteg:
  - Invisibilidad: Alta
  - Capacidad de guardar información: Media
  - Resistencia ante ataques estadísticos: Baja
  - Archivos no sospechosos: Las imágenes JPEG son las más extendidas en internet, por lo que no levanta sospechas
- Matrix Embedding:
  - Invisibilidad: Depende de si se utiliza para imágenes GIF o imágenes JPEG
  - Capacidad de guardar información: Baja
  - Resistencia ante ataques estadísticos: Alta
  - Archivos no sospechosos: Alta, ya que se puede usar tanto para imágenes GIF, como para imágenes JPEG.

# CAPÍTULO 3

# Diseño de la aplicación

En este capítulo, hablaremos de la implementación en Haskell de distintos algoritmos de esteganografía en imágenes. En nuestro caso, los algoritmos que vamos a implementar son LSB-replacement, LSB-matching y Matrix Embedding para imágenes GIF. Para tratar con dicho tipo de imágenes, usaremos la librería Juicy Pixels.

## 3.1– LSB-replacement

A lo largo de esta sección, se verán las distintas librerías y funciones que se han utilizado para implementar el algoritmo LSB-replacement en Haskell. Los pasos a seguir para aplicar este algoritmo son los siguientes:

1. Obtener el vector con las codificaciones de colores en formato RGB y expresar dichos valores numéricos en binario.
2. Expresar el mensaje que se quiere esconder en binario.
3. Sustituir el bit menos significativo de las codificaciones obtenidas en el paso 1 por un bit del mensaje.
4. Expresar las nuevas codificaciones en el sistema decimal y utilizar el nuevo vector para generar la imagen stego.

### 3.1.1. Librerías

Empezamos importando las siguientes librerías:

1. La librería `Data.Char`, que permite trabajar con los caracteres que forman los mensajes.
2. El vector de colores que se obtiene de las imágenes es de tipo `Data.Vector.Storable`, por lo que importamos dicha librería.
3. La librería `Data.Vector`, ya que se obtendrán las codificaciones binarias de los colores en forma de lista, y el tipo del vector de colores `Data.Vector.Storable` no admite almacenar listas.

4. Una librería que permita obtener los datos de una imagen GIF, que en nuestro caso será `Codec.Picture.Gif`

5. Para poder almacenar una imagen en formato GIF, importamos la librería `Codec.Picture.Saving`.

6. La librería `Codec.Picture.Types`, que contiene constructores que recogen información sobre los píxeles de una imagen.

7. Con el objetivo de usar funciones para leer un fichero y para almacenar el resultado de aplicarle el algoritmo a la imagen en un fichero, importamos las librerías `Data.ByteString` y `Data.ByteString.Lazy`.

Código 3.1: Librerías para el algoritmo LSB-replacement

```
1 import Data.Char
2 import qualified Data.Vector.Storable as VS
3 import qualified Data.Vector as V
4 import Codec.Picture.Gif
5 import Codec.Picture.Saving
6 import Codec.Picture.Types
7 import qualified Data.ByteString as BS (readFile)
8 import qualified Data.ByteString.Lazy as BSL (writeFile)
```

Código 3.1: Librerías para el algoritmo LSB-replacement

### 3.1.2. Funciones de codificación y decodificación

En primer lugar, son necesarias funciones que convierten el mensaje y el vector con la información de los colores de la imagen en binario para poder insertar en el bit menos significativo, un bit del mensaje. A su vez, también se necesitará otra función que convierta el vector con las codificaciones de los colores de cada píxel de la imagen en base 2 a un vector con dichas codificaciones en sistema decimal.

Se definen las funciones que permiten expresar un número decimal en uno en base 2 y viceversa:

Código 3.2: Función `int2bin`

```
1 int2bin :: Integral a => a -> [a]
2 int2bin n=
3   if n < 2
4   then [n]
5   else int2bin (n `div` 2)++[ n `mod` 2 ]
```

Código 3.2: Función `int2bin`

Código 3.3: Función `bin2int`

```
1 b2int :: Num p => [p] -> p
2 b2int []= 0
3 b2int (x:xs) = x + 2 * (b2int xs)
4 --Como tenemos que aplicar lo anterior en orden inverso
5 bin2int :: Num p => [p] -> p
```

```
bin2int (xs) =
  b2int (reverse xs)
```

Código 3.3: Función `bin2int`

A continuación, se tiene la función `bit8`, que da el octeto correspondiente a la lista de bits, añadiendo los ceros a la izquierda necesarios a la lista para que ésta tenga longitud ocho:

Código 3.4: Función `bit8`

```
bit8 :: Num a => [a] -> [a]
bit8 xs =
  if length ( xs)==8
  then xs
  else bit8(0:xs)
```

Código 3.4: Función `bit8`

Ahora se puede definir la función que convierte un mensaje a binario. Para ello, se utiliza la función `ord`, de la librería `Data.Char`, que devuelve el valor numérico asociado a un carácter:

Código 3.5: Función `menbin`

```
menbin :: [Char] -> [Int]
menbin xs=
  let ys= map int2bin (map ord xs)
  in (concatMap bit8 ys) ++[0,0,0,0,0,0,0,0]
```

Código 3.5: Función `menbin`

Como se puede observar, primero se ha expresado en binario los valores numéricos del mensaje xs y, después, se han expresado las listas correspondientes en octetos y añadido una lista de ceros para que sea más fácil extraer el mensaje de la imagen stego.

También se define la función `vecbin`, que convierte los valores numéricos de los colores en binario:

Código 3.6: Función `vecbin`

```
vecbin :: (VS.Storable a, Integral a) => VS.Vector a -> V.Vector [Int]
vecbin v =
  let n = VS.length v
  in V.generate n (\ i -> bit8 (int2bin (fromIntegral (v VS.! i))))
```

Código 3.6: Función `vecbin`

En esta función, se recibe un vector de tipo `Data.Vector.Storable` y se devuelve un vector de tipo `Data.Vector`, ya que los vectores del primer tipo no admiten que se almacenen listas (las codificaciones binarias de los colores). Para conseguir un vector de tipo `Data.Vector`, se utiliza la función `generate` de dicha librería.

Lo último que queda es la función que, una vez sustituido el bit menos significativo de cada número por un bit del mensaje, devuelve las codificaciones en binario al sistema decimal:

Código 3.7: Función `binvect`

```
binvect :: (VS.Storable a, Integral a) => V.Vector [Int] -> VS.Vector a
binvect v =
  let n = V.length v
  in VS.generate n (\ i -> fromIntegral (bin2int (v V.! i)))
```

Código 3.7: Función `binvect`

Aquí, se utiliza `fromIntegral` para transformar los datos de `Int` a `GHC.Word.Word8`. También se usa la función `generate` pero, esta vez, de la librería `Data.Vector.Storable`, para obtener un vector de ese tipo.

### 3.1.3. Inserción del mensaje mediante el algoritmo LSB-replacement

El objetivo al que se quiere llegar es una función que reciba el mensaje y el vector de la codificación de los colores que extrae del tipo `Image` y devuelva otro vector al que se le ha aplicado el algoritmo LSB-replacement.

Para ello, se necesita una función que permita introducir el mensaje en la codificación binaria de los colores de los píxeles mediante este algoritmo. Primero, se define la función `sustituirlsb`, que recibe dos listas de bits, una correspondiente a la codificación de un color en uno de los píxeles de la imagen y otra correspondiente al mensaje en binario, y sustituye el bit menos significativo de la primera lista (el octavo bit) por el primer bit del mensaje:

Código 3.8: Función `sustituirlsb`

```
sustituirlsb :: [a] -> [a] -> [a]
sustituirlsb xs [] = xs++[]
sustituirlsb xs (m:ms)=
  (init xs)++[m]
```

Código 3.8: Función `sustituirlsb`

A continuación, se tiene la función que cambia cada número binario, correspondiente a la codificación de cada uno de los colores de cada píxel, por el resultante al sustituir el bit menos significativo por un bit del mensaje:

Código 3.9: Función `insMensaje`

```
insMensaje :: V.Vector [a] -> [a] -> V.Vector [a]
insMensaje pxs ms=
  let n= V.length pxs
  in V.generate n (\i-> sustituirlsb (pxs V.! i) (drop i ms))
```

Código 3.9: Función `insMensaje`

Obsérvese que conforme se inserta uno de los bits del mensaje, este bit se desecha, de manera que se va cogiendo cada uno de los bits que forma el mensaje. Una vez se acaba el mensaje, se reduce al caso base, que en este caso es no modificar nada.

Ya se puede definir la función para insertar un mensaje en el vector de colores de una imagen:

Código 3.10: Función `replace`

```
replace :: (VS.Storable a1, VS.Storable a2, Integral a1, Integral a2) =>
    VS.Vector a2 -> [Char] -> VS.Vector a1
replace v ms=
  binvect (insMensaje (vecbin v) (menbin ms))
```

Código 3.10: Función `replace`

### 3.1.4. Modificación de la imagen

Ahora que tenemos las funciones auxiliares para aplicar el algoritmo, necesitamos cargar una imagen GIF, aplicarle el algoritmo y descargarnos la imagen stego:

Código 3.11: Acción `alglsbreplacement`

```
alglsbreplacement :: FilePath -> [Char] -> FilePath -> IO ()
alglsbreplacement im ms st = do
  byteString <- BS.readFile im --(1)
  let (Right dynamicImage1) = decodeGif byteString --(2)
  let (ImageRGB8 imagen1) = dynamicImage1 --(3)
  let (Image w h pxs) = imagen1 --(4)
  let rxs = replace pxs ms --(5)
  let(Right byteString2) =imageToGif (ImageRGB8(Image w h rxs)) --(6)
  BSL.writeFile st byteString2 --(7)
```

Código 3.11: Acción `alglsbreplacement`

Como se puede observar, se trata de una acción, no una función, ya que tiene elementos de la mónada de entrada y salida `IO`. Esta acción recibe los datos `im`, que es la ruta de la imagen cover, `ms`, que es el mensaje que se quiere esconder en la imagen (expresado como una cadena de caracteres), y `st`, que es la ruta donde se va a guardar la imagen stego.

Expliquemos el código:

1. Se utiliza `BS.readFile`, para leer el contenido del fichero `im` y lo devuelve como un `IO Data.ByteString.Internal.ByteString`. Se recoge el dato generado por dicha acción, utilizando la asignación `<-`. Este dato es un `ByteString`, que se decodificará a continuación.

2. Se decodifica el `ByteString` obtenido en el paso anterior usando `decodeGif` para poder obtener la información de la imagen GIF. El valor devuelto se asigna con `let ..=...` y se obtiene un dato de tipo `Either String DynamicImage`. Este tipo agrupa los tipos `String` y `DynamicImage` en uno solo. Con los constructores `Left` y `Right` se pueden obtener uno de los dos tipos. Para obtener un dato del primer tipo (`String`), se tendría que usar `Left`, y para obtener un dato del segundo tipo (`DynamicImage`), se usaría `Right`. En este caso, el objetivo es obtener un dato del segundo tipo, por lo que se usa `Right`.

3. El tipo `DynamicImage` agrupa a los distintos tipos de píxeles. En este caso, se escoge el constructor `ImageRGB8`. El argumento de este constructor es de tipo `Image` y se asigna a la variable `imagen1`.

4. Los datos de tipo `Image` se construyen con el constructor `Image` con tres argumentos, la anchura medida en píxeles de la imagen (`w`), la altura medida en píxeles de la imagen (`h`), y un vector con la información de los colores de cada píxel (`pxs`). Este vector es el que se debe modificar para esconder el mensaje.

5. Se cambia el vector `pxs` de la imagen escondiendo el mensaje `ms` mediante el algoritmo LSB-replacement, utilizando la función `replace`. El nuevo vector se asigna a la variable `rxs`

6. En este paso, se almacena `(Image w h rxs)` como una `DynamicImage` con el constructor `ImageRGB8`. A continuación, usando la función `imageToGif`, se transforma la `DynamicImage` en una imagen en formato GIF, que tiene el tipo `Either String Data.ByteString.Lazy.Internal.ByteString`. Como sólo nos interesa la parte derecha, se utiliza el constructor `Right`.

7. Por último, `BSL.writeFile` almacena `byteString2` en un fichero.

## 3.2– LSB-matching

El algoritmo LSB-matching es muy parecido al algoritmo LSB-replacement. La diferencia entre ambos es que, en vez de sustituir el bit menos significativo por un bit del mensaje, en el algoritmo LSB-matching, cuando el bit menos significativo no coincide con el bit del mensaje, se suma 1 ó -1 aleatoriamente a la codificación binaria del color. Los pasos a seguir para aplicar este algoritmo son los siguientes:

1. Obtener el vector con las codificaciones de colores en formato RGB y expresar dichos valores numéricos en binario.

2. Expresar el mensaje que se quiere esconder en binario.

3. Generar una lista de números aleatorios.

4. Cambiar las codificaciones del paso 1 cuyo bit menos significativo no coincide con el bit del mensaje correspondiente. Para ello, se suma 1 ó -1 aleatoriamente.

5. Expresar las nuevas codificaciones en el sistema decimal y utilizar el nuevo vector para generar la imagen stego.

### 3.2.1. Librerías

Las librerías que se necesitan para implementar el algoritmo LSB-matching son las siguientes:

Código 3.12: Librerías LSB-matching

```
1 import Data.Char
2 import qualified Data.Vector as V
3 import qualified Data.Vector.Storable as VS
4 import Codec.Picture.Gif
5 import Codec.Picture.Saving
6 import Codec.Picture.Types
7 import qualified Data.ByteString as BS (readFile)
8 import qualified Data.ByteString.Lazy as BSL (writeFile)
```

```
9 import System.Random
```

Código 3.12: Librerías LSB-matching

Cómo se puede observar, la mayoría de las librerías son las mismas que para el algoritmo LSB-replacement, ya que todavía es necesario trabajar con los caracteres de los mensajes, los vectores de color de las imágenes tanto con las codificaciones en sistema decimal como en binario, y librerías para crear y extraer información de imágenes en formato GIF. La única librería que no se usaba antes es la librería `System.Random`. Esta librería permitirá generar números aleatorios.

### 3.2.2. Inserción del mensaje mediante el algoritmo LSB-matching

En primer lugar, se tiene que para realizar cambios se suma 1 ó -1, por lo que es necesaria una función que permita hacer esta suma. La función es la siguiente:

Código 3.13: Función `sumBinarioUno`

```
1  sumBinarioUno :: (Eq t, Num t) => [t] -> t -> [t]
2  sumBinarioUno ns 0 = ns
3  sumBinarioUno ns 1 =
4      if all (== 1) ns
5      then sumBinarioUno ns (-1)
6      else if last ns == 1
7           then (sumBinarioUno (init ns) 1) ++[0]
8           else (sumBinarioUno (init ns) 0) ++[last ns + 1]
9  sumBinarioUno ns (-1) =
10     if all (== 0) ns
11     then sumBinarioUno ns 1
12     else if last ns == 0
13          then (sumBinarioUno (init ns) (-1)) ++[1]
14          else (sumBinarioUno (init ns) 0) ++[last ns - 1]
```

Código 3.13: Función `sumBinarioUno`

Como se puede observar, es necesario distinguir los casos en los que el número binario es [1,1,1,1,1,1,1,1] y [0,0,0,0,0,0,0,0]. El problema con estos casos es que si hay que sumar 1 a [1,1,1,1,1,1,1,1], se debería obtener [1,0,0,0,0,0,0,0,0], y si se suma -1 a [0,0,0,0,0,0,0,0], se debería obtener [0,1,1,1,1,1,1,1,1], es decir, se tendría que añadir un elemento más a la lista. Como añadir un elemento a la lista no está contemplado en esta función, si se tiene que todos los elementos de la lista son 1, siempre se suma -1, y si todos son 0, siempre se suma 1.

También se necesita generar una lista cuyos elementos son 1 ó -1 de manera aleatoria. Para ello, se utiliza una acción, ya que una función da los mismos resultados si se tienen los mismos datos. Con la acción `randomRIO(0,1):: IO Int`, se obtiene un número aleatorio que puede ser 0 ó 1. Además, con `sequence` se puede obtener una lista de números aleatorios. Como lo que interesa es que los números aleatorios sean 1 ó -1 (ya que son los números que vamos a sumar para cambiar el bit menos significativo), se define una función que cambie un 0 por un -1:

Código 3.14: Función `cambiarCeros`

```
cambiarCeros :: (Eq a, Num a, Num p) => a -> p
cambiarCeros n=
  if n==0
  then -1
  else 1
```

Código 3.14: Función `cambiarCeros`

A continuación, se tiene la acción que devuelve una lista cuyos elementos son 1 ó -1 de forma aleatoria:

Código 3.15: Función `randomNumbers`

```
randomNumbers :: Int -> IO [Int]
randomNumbers n = do
  ms <- sequence [randomRIO (0,1) :: IO Int | k <- [1..n]]
  let ns = map cambiarCeros ms
  return ns
```

Código 3.15: Función `randomNumbers`

Ya se tiene una manera de generar una lista de números aleatorios. Ahora, queda definir funciones que permitan modificar la codificación de colores de los píxeles según el algoritmo LSB-matching para esconder un mensaje.

En primer lugar, se tiene la función `match`, que hace los cambios necesarios en el número binario correspondiente a la codificación de un color para que el bit menos significativo coincida con el bit del mensaje:

Código 3.16: Función `match`

```
match :: (Eq t, Num t) => [t] -> [t] -> [t] -> [t]
match xs [] [] = xs
match xs (m:ms) (n:ns)=
  if last xs ==m
  then xs
  else sumBinarioUno xs n
```

Código 3.16: Función `match`

Esta función, en caso de que el bit menos significativo sea igual que el bit del mensaje, no cambia nada, y, en caso contrario, le suma a la codificación en binario del color 1 ó -1 aleatoriamente.

Ahora, se define la función que cambia el vector cuyos elementos son las listas que corresponden a la codificación de los colores de la imagen, por el vector resultante al aplicarle la función anterior a cada lista, dados el mensaje en binario y la lista de números aleatorios:

Código 3.17: Función `insMatching`

```
insMatching :: (Eq t, Num t) => V.Vector [t] -> [t] -> [t] -> V.Vector [t]
insMatching pxs ms ns=
  let n= V.length pxs
  in V.generate n (\i-> match (pxs V.! i) (drop i ms) (drop i ns))
```

Código 3.17: Función `insMatching`

Por último, queda la función que recibe el vector de las codificaciones en sistema decimal de los colores de cada píxel de la imagen, la cadena de caracteres que forman el mensaje y una lista de números aleatorios, y devuelve el vector de las codificaciones en sistema decimal de los colores de cada píxel de la imagen stego:

Código 3.18: Función `lsbMatching`

```
1 lsbMatching :: (VS.Storable a1, VS.Storable a2, Integral a1, Integral a2) =>
2     VS.Vector a2 -> [Char] -> [Int] -> VS.Vector a1
3 lsbMatching v ms ns =
4   binvect (insMatching (vecbin v) (menbin ms) ns)
```

Código 3.18: Función `lsbMatching`

Como se puede observar, se han utilizado las funciones `menbin` (código 3.5), para expresar el mensaje en binario, `vecbin` (código 3.6), para obtener la codificación de los colores en binario, y `binvect` (código 3.7), para devolver un vector de tipo `Data.Vector.Storable` que se pueda usar de tercer argumento para el constructor `Image` para la imagen stego, que se encuentran definidas en la sección anterior.

### 3.2.3. Modificación de la imagen

Una vez definidas las funciones que permiten esconder un mensaje en una imagen mediante el algoritmo LSB-matching, queda definir una acción similar a la presentada en el código 3.11, para extraer la información de un fichero, modificarla, y guardarla en un nuevo fichero:

Código 3.19: Acción `algLsbMatching`

```
1  algLsbMatching :: FilePath -> [Char] -> FilePath -> IO ()
2  algLsbMatching im ms st = do
3    byteString1 <- BS.readFile im
4    let (Right dynamicImage1) = decodeGif byteString1
5    let (ImageRGB8 imagen1) = dynamicImage1
6    let (Image w h pxs) = imagen1
7    ns <- randomNumbers ((Prelude.length ms)*8 +8)
8    let qxs = lsbMatching pxs ms ns
9    let (Right byteString2) = imageToGif (ImageRGB8 (Image w h qxs))
10   BSL.writeFile st byteString2
```

Código 3.19: Acción `algLsbMatching`

Veamos las diferencias con el código de `alglsbreplacement` (código 3.11):

1. En la línea 7 se guarda una lista de números aleatorios (1 ó -1) en la variable `ns` mediante la asignación `<-`. Esta lista es de longitud el tamaño del mensaje en binario más ocho (que corresponde a la lista de ceros que se añade al final del mensaje ), de manera que a cada bit que se esconde, le corresponde un número aleatorio de la lista.

2. En la línea 9, en vez de aplicar la función `lsb` se aplica `lsbMatching`, para esconder el mensaje usando el algoritmo LSB-Matching.

## 3.3– Matrix Embedding

En esta sección, se verá cómo implementar este algoritmo en Haskell para esconder 2, 3 ó 4 bits del mensaje por cada bloque de la imagen cover. Los pasos a seguir para aplicar este algoritmo son los siguientes:

1. Obtener el vector con las codificaciones de colores en formato RGB. Después, se escogen los elementos necesarios de este vector para esconder el mensaje, que se dividirá en bloques de 2,3 ó 4 bits. Al número de bits en cada bloque se le denominará $n$.

2. Dividir el vector con los elementos necesarios para la codificación en bloques de $2^n - 1$ elementos.

3. A cada par formado por un bloque del vector de las codificaciones de los colores de la imagen y un bloque del mensaje se le aplica los siguientes pasos:

   *a*) Extraer los bits menos significativos necesarios para esconder un bloque del mensaje y formar una matriz de una sola columna con ellos.

   *b*) Formar una matriz con los vectores no nulos de dimensión $n$

   *c*) Definir una función que compruebe si se cumple la ecuación correspondiente, explicada en el capítulo 2.

   *d*) Definir una función que cambie el bit menos significativo.

   *e*) Dividir por casos en función de la longitud del mensaje que se quiera esconder y definir una función para cada caso que realice los cambios oportunos para esconder el mensaje, obteniendo un vector con las codificaciones nuevas de los colores en binario.

   *f*) Expresar estas codificaciones en el sistema decimal

4. Concatenar los vectores resultantes con el resto del vector de la codificación que no se ha usado y utilizar este vector como el vector de las codificaciones de colores para la imagen stego.

### 3.3.1. Librerías

Las librerías que se necesitan para este algoritmo son:

Código 3.20: Librerías para el algoritmo Matrix Embedding

```
1  import qualified Data.Vector as V
2  import qualified Data.Vector.Storable as VS
3  import Codec.Picture.Gif
4  import Codec.Picture.Saving
5  import Codec.Picture.Types
6  import qualified Data.ByteString as BS (readFile)
7  import qualified Data.ByteString.Lazy as BSL (writeFile)
8  import qualified Data.Matrix as M
9  import qualified Data.List as L
10 import Data.Char
```

Código 3.20: Librerías para el algoritmo Matrix Embedding

La mayoría de las librerías ya se han visto en las secciones anteriores. Las que se han añadido en este algoritmo son:

1. La librería `Data.Matrix`, para poder multiplicar matrices con el objetivo de comprobar que los bits menos significativos satisfacen las ecuaciones que tienen que cumplir.

2. La librería `Data.List`, para obtener los vectores no nulos de la longitud necesaria y poder formar una matriz a partir de ellos.

### 3.3.2. Modificación del bit menos significativo

En primer lugar, se necesita una función que compruebe si se cumplen las relaciones entre los bits de un bloque del mensaje y los bits menos significativos de un bloque de los píxeles de la imagen. Para ello, primero se define una función que permita obtener el vector con los bits menos significativos:

Código 3.21: Función `extraerBits`

```
extraerBits :: V.Vector [b] -> V.Vector b
extraerBits pxs =
  (V.map Prelude.last (pxs))
```

Código 3.21: Función `extraerBits`

Por otro lado, se define la función `vectoresNoNulos`. Esta función permite crear una matriz con los vectores no nulos de longitud n:

Código 3.22: Función `vectoresNoNulos`

```
vectoresNoNulos :: (Eq a, Num a) => Int -> [[a]]
vectoresNoNulos n=
 L.nub ( L.concatMap L.permutations ([(L.replicate i 1)L.++(L.replicate (n-i) 0)
     | i<-[1..n]]))
```

Código 3.22: Función `vectoresNoNulos`

En esta función se ha utilizado la función `replicate` y la concatenación de listas para conseguir listas en las que aumente el número de unos. Además, se ha utilizado la función `permutations` de la librería `Data.List` para obtener todas las permutaciones de estos vectores y la función `nub` para eliminar las repeticiones. Las matrices que se generarán a partir de estas listas para $n = 2, 3$ y 4 son las siguientes:

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Ya se puede definir la función `mensaje`, que comprueba si se satisfacen las relaciones entre los bits menos significativos y los bits del mensaje. Esta función recibe la matriz obtenida con la función `vectoresNoNulos` por el vector con los bits menos significativos de los píxeles en el vector con las codificaciones de los colores, y se comprueba si el n-ésimo elemento del resultado coincide con el n-ésimo bit del mensaje:

Código 3.23: Función `mensaje`

```
1 mensaje :: Integral a => M.Matrix a -> [a] -> Int -> Bool
2 mensaje m ms n =
3   (ms Prelude.!!(n-1))== (m M.! (n,1))`mod`2
```

Código 3.23: Función `mensaje`

Una vez detectado que no se cumple alguna condición, se tiene que cambiar el bit menos significativo correspondiente. Para ello, se define la siguiente función:

Código 3.24: Función `cambiolsb`

```
1 cambiolsb :: Integral a => [a] -> [a]
2 cambiolsb xs =
3   (Prelude.init xs)Prelude.++[((Prelude.last xs)+1)`mod`2]
```

Código 3.24: Función `cambiolsb`

Para localizar el bit menos significativo que hay que modificar y cambiarlo, se tiene la siguiente función:

Código 3.25: Función `insMatrix`

```
1 insMatrix :: Integral a => V.Vector [a] -> Int -> V.Vector [a]
2 insMatrix pxs m =
3   let n= V.length pxs
4   in V.generate n (\i-> if i==(m-1) then cambiolsb (pxs V.! i) else (pxs V.! i))
```

Código 3.25: Función `insMatrix`

### 3.3.3. Inserción del mensaje mediante el algoritmo Matrix Embedding

A continuación, se definen funciones para aplicar el algoritmo Matrix Embedding con un bloque del mensaje de 2, 3 ó 4 bits respectivamente. A lo largo de esta sección, se denotará $o_i$, como el i-ésimo bit menos significativo. Para dichas funciones, se utiliza `binvect` (código 3.7), para obtener la codificación de los colores de los píxeles en sistema decimal, después de haber aplicado el algoritmo. Además, se utilizan las funciones `vectoresNoNulos` (código 3.22) y `extraerBits` (código 3.21) para obtener la matriz (que en este caso es un vector) con la que hay que comparar el bloque del mensaje para comprobar que se cumplen las condiciones necesarias para esconder dicho bloque.

#### Bloques de 2 bits

Para este caso, por cada par $m_1m_2$ en el mensaje, donde $m_i$ es un bit, se busca que se satisfaga que $m_1 = o_1 \oplus o_3$ y $m_2 = o_2 \oplus o_3$. Teniendo en cuenta que "cumplir la condición $m_i$" se refiere a que se satisface la ecuación cuya parte izquierda es $m_i$, las modificaciones que se tendrían que realizar son las siguientes:

1. Si se cumplen las condiciones para $m_1$ y $m_2$, no hay que hacer ningún cambio.
2. Si no se cumple la condición para $m_1$, entonces se cambia $o_1$.
3. Si no se cumple la condición para $m_2$, entonces se cambia $o_2$.
4. Si no se cumplen ni $m_1$, ni $m_2$ se cambia $o_3$

Teniendo en cuenta estos casos, se define la siguiente función, que realiza el cambio oportuno en cada caso:

Código 3.26: Función `matrixEmbedding2`

```
matrixEmbedding2 :: (VS.Storable a, Integral a) =>V.Vector [Int] -> [Int] -> VS.
    Vector a
matrixEmbedding2 pxs ms =
  let m = M.transpose (M.fromLists (vectoresNoNulos 2))*M.colVector (extraerBits
      pxs)
  in if mensaje m ms 1
     then if mensaje m ms 2
          then binvect pxs
          else binvect (insMatrix pxs 2)
     else if mensaje m ms 2
          then binvect (insMatrix pxs 1)
          else binvect (insMatrix pxs 3)
```

Código 3.26: Función `matrixEmbedding2`

### Bloques de 3 bits

Se considera el bloque $m_1m_2m_3$, donde $m_i$ es un bit. Se busca que se satisfaga $m_1 = o_1 \oplus o_4 \oplus o_6 \oplus o_7$ , $m_2 = o_2 \oplus o_4 \oplus o_5 \oplus o_7$ y $m_3 = o_3 \oplus o_5 \oplus o_6 \oplus o_7$ . Teniendo en cuenta que "cumplir la condición $m_i$" se refiere a que se satisface la ecuación cuya parte izquierda es $m_i$, los cambios a realizar son los siguientes:

1. Si se cumplen las condiciones para $m_1$, $m_2$ y $m_3$, no hay que hacer ningún cambio.
2. Si no se cumple la condición para $m_1$, entonces se cambia $o_1$
3. Si no se cumple la condición para $m_2$, entonces se cambia $o_2$
4. Si no se cumple la condición $m_3$, se cambia $o_3$
5. Si se tiene que no es verdad ni $m_1$ ni $m_2$, se cambia $o_4$.
6. Si no es verdad ni $m_1$, ni $m_3$, se modificará el bit $o_6$
7. Si no se cumple ni $m_2$, ni $m_3$, se cambia $o_5$
8. Si no se cumple ninguna de las condiciones, se cambia $o_7$

Por lo tanto, la función para esconder un mensaje de 3 bits en una imagen mediante el algoritmo Matrix Embedding sería la siguiente:

Código 3.27: Función `matrixEmbedding3`

```
matrixEmbedding3 :: (VS.Storable a, Integral a) =>V.Vector [Int] -> [Int] -> VS.
    Vector a
matrixEmbedding3 pxs ms =
  let m = M.transpose (M.fromLists (vectoresNoNulos 3))*M.colVector (extraerBits
      pxs)
  in if mensaje m ms 1
     then if mensaje m ms 2
          then if mensaje m ms 3
               then binvect pxs
               else binvect (insMatrix pxs 3)
          else if mensaje m ms 3
               then binvect (insMatrix pxs 2)
               else binvect (insMatrix pxs 5)
     else if mensaje m ms 2
          then if mensaje m ms 3
               then binvect (insMatrix pxs 1)
               else binvect (insMatrix pxs 6)
          else if mensaje m ms 3
               then binvect (insMatrix pxs 4)
               else binvect (insMatrix pxs 7)
```

Código 3.27: Función `matrixEmbedding3`

### 3.3.4. Bloques de 4 bits

Ahora, se tiene que considera el bloque de la forma $m_1m_2m_3m_4$, donde $m_i$ es un bit. Se busca que estos bits cumplan que $m_1 = o_1 \oplus o_5 \oplus o_7 \oplus o_{10} \oplus o_{11} \oplus o_{13} \oplus o_{14} \oplus o_{15}$, $m_2 = o_2 \oplus o_5 \oplus o_6 \oplus o_9 \oplus o_{11} \oplus o_{12} \oplus o_{14} \oplus o_{15}$, $m_3 = o_3 \oplus o_6 \oplus o_7 \oplus o_8 \oplus o_{11} \oplus o_{12} \oplus o_{13} \oplus o_{15}$ , y $m_4 = o_4 \oplus o_8 \oplus o_9 \oplus o_{10} \oplus o_{12} \oplus o_{13} \oplus o_{14} \oplus o_{15}$. Para que se verifiquen estas condiciones, habrá que realizar los siguientes cambios:

1. Si se cumplen las condiciones para $m_1$, $m_2$ y $m_3$, no hay que hacer ningún cambio.
2. Si no se cumple la condición para $m_1$, entonces se cambia $o_1$
3. Si no se cumple la condición para $m_2$, entonces se cambia $o_2$
4. Si no se cumple la condición $m_3$, se cambia $o_3$
5. Si no se cumple la condición $m_4$, se cambia $o_4$
6. Si se tiene que no es verdad ni $m_1$ ni $m_2$, se cambia $o_5$.
7. Si no es verdad ni $m_1$, ni $m_3$, se modificará el bit $o_7$
8. Si no se cumple la condición $m_1$, ni $m_4$, se modificará el bit $o_{10}$
9. Si no se cumple la condición $m_2$, ni $m_3$, se modificará el bit $o_6$
10. Si no se cumple la condición $m_2$, ni $m_4$, se modificará el bit $o_9$
11. Si no se cumple la condición $m_3$, ni $m_4$, se modificará el bit $o_8$

12. Si no se cumplen las condiciones $m_1$, $m_2$ y $m_3$, se modificará el bit $o_{11}$

13. Si no se cumplen las condiciones $m_1$, $m_2$ y $m_4$, se modificará el bit $o_{14}$

14. Si no se cumplen las condiciones $m_1$, $m_3$ y $m_4$, se modificará el bit $o_{13}$

15. Si no se cumplen las condiciones $m_2$, $m_3$ y $m_4$, se modificará el bit $o_{12}$

16. Si no se cumple ninguna de las ecuaciones, se modificará el bit $o_{15}$

Teniendo en cuenta las modificaciones anteriores, se define la siguiente función para poder realizarlas:

Código 3.28: Función `matrixEmbedding4`

```
matrixEmbedding4 :: (VS.Storable a, Integral a) =>V.Vector [Int] -> [Int] -> VS.
    Vector a
matrixEmbedding4 pxs ms=
 let m = M.transpose (M.fromLists (vectoresNoNulos 4))*M.colVector (extraerBits
     pxs)
 in  if mensaje m ms 1
     then if mensaje m ms 2
          then if mensaje m ms 3
               then if mensaje m ms 4
                    then  binvect pxs
                    else binvect (insMatrix pxs 4)
               else if mensaje m ms 4
                    then binvect (insMatrix pxs 3)
                    else binvect (insMatrix pxs 8)
          else if mensaje m ms 3
               then if mensaje m ms 4
                    then binvect (insMatrix pxs 2)
                    else binvect (insMatrix pxs 9)
               else if mensaje m ms 4
                    then binvect (insMatrix pxs 6)
                    else binvect (insMatrix pxs 12)
     else if mensaje m ms 2
          then if mensaje m ms 3
               then if mensaje m ms 4
                    then binvect (insMatrix pxs 1)
                    else binvect (insMatrix pxs 10)
               else if mensaje m ms 4
                    then binvect (insMatrix pxs 7)
                    else binvect (insMatrix pxs 13)
          else if mensaje m ms 3
               then if mensaje m ms 4
                    then binvect (insMatrix pxs 5)
                    else binvect (insMatrix pxs 14)
               else if mensaje m ms 4
                    then binvect (insMatrix pxs 11)
                    else binvect (insMatrix pxs 15)
```

Código 3.28: Función `matrixEmbedding4`

### 3.3.5. División en bloques y concatenación

Ahora que se han definido funciones para cada bloque del mensaje, falta definir funciones que permitan dividir el vector con las codificaciones y el mensaje en bloques de $2^n - 1$ elementos y $n$ bits respectivamente.

La función para dividir en bloques el vector con las codificaciones binarias se define de la siguiente manera:

Código 3.29: Función `dividirbloques`

```
1 dividirbloques :: V.Vector a -> Int -> [V.Vector a]
2 dividirbloques pxs n =
3   let z= div (length pxs) n
4   in [V.take n (V.drop (n*i) pxs)| i<-[0..z-1]]
```

Código 3.29: Función `dividirbloques`

En esta función, como se verá más adelante, `pxs` corresponde a la codificación binaria de los colores de la imagen, mientras que `n` es el tamaño del bloque.

Para dividir la codificación binaria del mensaje en bloques de 2,3 ó 4 bits se utilizan las siguientes funciones:

Código 3.30: Función `men2bits`

```
1 men2bits :: [a] -> [[a]]
2 men2bits []=[]
3 men2bits xs = take 2 xs:(men2bits (drop 2 xs))
```

Código 3.30: Función `men2bits`

Código 3.31: Función `men3bits`

```
1 men3bits :: [a] -> [[a]]
2 men3bits []=[]
3 men3bits xs = take 3 xs:(men3bits (drop 3 xs))
```

Código 3.31: Función `men3bits`

Código 3.32: Función `men4bits`

```
1 men4bits :: [a] -> [[a]]
2 men4bits []=[]
3 men4bits xs = take 4 xs:(men4bits (drop 4 xs))
```

Código 3.32: Función `men4bits`

Ahora, se definen las funciones `embedding4`, `embedding3` y `embedding2`, que aplican el algoritmo Matrix Embedding a cada bloque dependiendo de su tamaño:

Código 3.33: Función `embedding4`

```
1 embedding4 :: (VS.Storable a, Integral a) =>V.Vector [Int] -> [[Int]] -> [VS.
    Vector a]
2 embedding4 pxs ms =
3   let bloques = dividirbloques pxs (2^4-1)
4       m = length bloques
```

```
5   in [matrixEmbedding4 (bloques Prelude.!! i) (ms Prelude.!! i) |
6       i <- [0..m-1]]
```

Código 3.33: Función `embedding4`

Código 3.34: Función `embedding3`

```
1 embedding3 :: (VS.Storable a, Integral a) =>V.Vector [Int] -> [[Int]] -> [VS.
      Vector a]
2 embedding3 pxs ms =
3   let bloques = dividirbloques pxs (2^3-1)
4       m = length bloques
5   in [matrixEmbedding3 (bloques Prelude.!! i) (ms Prelude.!! i) |
6       i <- [0..m-1]]
```

Código 3.34: Función `embedding3`

Código 3.35: Función `embedding2`

```
1 embedding2 :: (VS.Storable a, Integral a) =>V.Vector [Int] -> [[Int]] -> [VS.
      Vector a]
2 embedding2 pxs ms =
3   let bloques = dividirbloques pxs (2^2-1)
4       m = length bloques
5   in [matrixEmbedding2 (bloques Prelude.!! i) (ms Prelude.!! i) |
6       i <- [0..m-1]]
```

Código 3.35: Función `embedding2`

La función `embedding4` (código 3.33) se usa para cuando el mensaje se divide en bloques de 4 bits, la función `embedding3` (código 3.34) para cuando el mensaje se divide en bloques de 3 bits y `embedding2` (código 3.35) para cuando se divida en bloques de 2 bits.

Ahora, se define la función que recibe el vector con las codificaciones de los colores de la imagen cover, el mensaje y el tamaño de los bloques en los que se va a dividir el mensaje y devuelve el vector con las codificaciones de los píxeles de la imagen stego:

Código 3.36: Función `vectorStegoMatrix`

```
1  vectorStegoMatrix :: (VS.Storable a, Integral a, Integral b) => VS.Vector a -> [
       Char] -> b -> VS.Vector a
2  vectorStegoMatrix pxs ms n =
3          if n==4
4          then let m=((Prelude.length (men4bits (menbin ms))))*(2^4 -1)
5               in VS.concat ((embedding4 (vecbin(VS.take m pxs)) (men4bits(menbin
                       ms)))++ [VS.drop m pxs])
6          else if n==3
7               then let m=((Prelude.length (men3bits (menbin ms))))*(2^3 -1)
8                    in VS.concat ((embedding3 (vecbin(VS.take m pxs)) (men3bits(
                           menbin ms) ))++ [VS.drop m pxs])
9               else let m=((Prelude.length (men2bits (menbin ms))))*(2^2 -1)
10                   in VS.concat ((embedding2 (vecbin(VS.take m pxs)) (men2bits(
                           menbin ms) ) )++ [VS.drop m pxs])
```

Código 3.36: Función `vectorStegoMatrix`

Nótese que la parte del vector de las codificaciones que no se usa para esconder el mensaje no se modifica. Además, se usan las funciones `vecbin` y `menbin` para obtener las codificaciones binarias del vector de colores y del mensaje.

### 3.3.6. Modificación de la imagen

A continuación, se incluye el código que define una acción que permite seleccionar un fichero, esconder un mensaje mediante el algoritmo Matrix Embedding, y guardar la imagen stego en un nuevo fichero:

Código 3.37: Acción `algMatrixEmbedding`

```
1  algMatrixEmbedding :: FilePath -> [Char] -> FilePath -> IO ()
2  algMatrixEmbedding im ms st= do
3    byteString1 <- BS.readFile im
4    let (Right dynamicImage1) = decodeGif byteString1
5    let (ImageRGB8 imagen1) = dynamicImage1
6    let (Image w h pxs) = imagen1
7    let n= VS.length pxs
8    let s= Prelude.length (menbin ms)
9    if  ((Prelude.length (men4bits (menbin ms))))*(2^4 -1)<=n
10   then do putStrLn ("Mensaje dividido en bloques de 4")
11           let rxs =vectorStegoMatrix pxs ms 4
12           let(Right byteString2) =imageToGif (ImageRGB8(Image w h rxs))
13           BSL.writeFile st byteString2
14   else if (mod s 3==0) && (((Prelude.length (men3bits(menbin ms) ))*(2^3 -1))<=n)
15        then do putStrLn ("Mensaje dividido en bloques de 3")
16                let rxs =vectorStegoMatrix pxs ms 3
17                let(Right byteString2) =imageToGif (ImageRGB8(Image w h rxs))
18                BSL.writeFile st byteString2
19        else if ((Prelude.length (men2bits (menbin ms) ))*(2^3 -1))<=n
20             then do putStrLn ("Mensaje dividido en bloques de 2")
21                     let rxs= vectorStegoMatrix pxs ms 2
22                     let(Right byteString2) =imageToGif (ImageRGB8(Image w h rxs))
23                     BSL.writeFile st byteString2
24             else putStrLn ("ERROR: Mensaje muy largo. No se pudo esconder el
                   mensaje")
```

Código 3.37: Acción `algMatrixEmbedding`

Este código se diferencia de los códigos de `alglsbreplacement` ( código 3.11) y de `algLsbMatching` (código 3.19) en los siguientes puntos:

- En las líneas 7 y 8 se obtienen las longitudes del vector con las codificaciones de los píxeles de la imagen y de la codificación binaria del mensaje.
- A partir de la línea 9, se distinguen los siguientes casos en función de la longitud del mensaje y el tamaño de la imagen (nótese que las dos últimas líneas en cada caso, menos el último, son para guardar la imagen stego obtenida):
  - Si el tamaño de la imagen es suficiente como para esconder el mensaje dividiéndolo en bloques de 4 bits, se emite un mensaje que lo indique y se aplica la función `vectorStegoMatrix` (código 3.36) para bloques de 4 bits.

- Si el tamaño de la imagen es suficiente como para esconder el mensaje dividiéndolo en bloques de 3 bits y su longitud es divisible por 3, se emite un mensaje que indique que el mensaje se dividirá en bloques de 3 bits y se aplica la función `vectorStegoMatrix` (código 3.36) para bloques de 3 bits.
- Si el tamaño de la imagen es suficiente como para esconder el mensaje dividiéndolo en bloques de 2 bits, se emite un mensaje que lo indique y se aplica la función `vectorStegoMatrix` (código 3.36) para bloques de 2 bits.
- En caso contrario, se indica que el mensaje es muy largo y no se aplica el algoritmo.

## 3.4– Extracción del mensaje

Esta sección se va a centrar en la implementación en Haskell de algoritmos que permitan extraer los mensajes escondidos con los algoritmos esteganográficos tratados en este capítulo.

### 3.4.1. Algoritmos LSB-replacement y LSB-matching

Los algoritmos LSB-replacement y LSB-matching tienen en común que, en la codificación binaria de los colores de los píxeles, los bits del mensaje se encuentran en el bit menos significativo, por lo que se puede usar el mismo algoritmo para extraer el mensaje escondido usando estos algoritmos. A continuación, se explica cómo se ha implementado este algoritmo. El esquema que se va a seguir es el siguiente:

1. Se extraen los bits menos significativos del vector con las codificaciones binarias de los colores de la imagen stego.
2. Se seleccionan los bits pertenecientes al mensaje.
3. Se obtienen los caracteres del mensaje a partir de su codificación binaria.

#### Librerías

Para implementar el algoritmo de extracción, se importan las siguientes librerías:

Código 3.38: `Librerías para el algoritmo de extracción del mensaje`

```
1 import Data.Char
2 import qualified Data.Vector.Storable as VS
3 import qualified Data.Vector as V
4 import Codec.Picture.Gif
5 import Codec.Picture.Types
6 import qualified Data.ByteString as BS (readFile)
```

Código 3.38: `Librerías para el algoritmo de extracción del mensaje`

En este caso, no se necesitan las librerías que se utilizaban en los algoritmos de inserción para guardar la imagen stego en un fichero, porque lo que se obtiene en el algoritmo es un mensaje. Además, esta vez se utiliza la librería `Data.Char` para obtener los caracteres del mensaje. Las otras librerías tienen la misma función que en la implementación de los algoritmos de codificación.

### Algoritmo para extraer el mensaje

En primer lugar, se necesita una función que extraiga los bits menos significativos del vector, de tipo `Data.Vector`, con la codificación binaria de los colores de la imagen stego:

Código 3.39: Función `extraerLSB`

```
extraerLSB :: (VS.Storable a, Integral a) => VS.Vector a -> [Int]
extraerLSB v =
 (V.toList ( V.map last (vecbin v)))
```

Código 3.39: Función `extraerLSB`

Ahora, para extraer los valores asociados a los caracteres del mensaje, se define la función `menint`. Par ello, primero se utiliza una función auxiliar que divida los bits menos significativos obtenidos anteriormente en grupos de ocho:

Código 3.40: Función `menintaux`

```
menintaux :: [a] -> [[a]]
menintaux xs=
  let n =length xs
  in [take 8 (drop i xs) | i<-[0,8..n]]
```

Código 3.40: Función `menintaux`

Se sabe, por la definición de la función `menbin` (código 3.5), que el mensaje escondido en binario termina con una lista de ceros. Usando esta información se define otra función auxiliar que concatene los elementos de la lista obtenida con la función anterior y que, cuando llegue a la lista formada por ceros, se detenga:

Código 3.41: Función `menintaux2`

```
menintaux2 :: (Eq a, Num a) => [[a]] -> [[a]]
menintaux2 [] = []
menintaux2 (x:xs)=
  if x== [0,0,0,0,0,0,0,0]
  then []
  else x:menintaux2 xs
```

Código 3.41: Función `menintaux2`

Por lo tanto, con esta función, se puede extraer el mensaje en binario. A continuación, se usan estas funciones auxiliares y la función `bin2int` (código 3.3) para obtener los dígitos que corresponden al mensaje:

Código 3.42: Función `menint`

```
menint :: (Num b, Eq b) => [b] -> [b]
menint xs = map bin2int ( menintaux2 (menintaux xs))
```

Código 3.42: Función `menint`

Finalmente, se define una función que devuelva los caracteres del mensaje:

Código 3.43: Función `listaCarac`

```
listaCarac :: [Int] -> [Char]
```

```
2 listaCarac xs =
3   map chr xs
```

Código 3.43: Función `listaCarac`

A continuación, se encuentra el código para obtener el mensaje, escondido con el algoritmo LSB-replacement o LSB-matching, de una imagen stego:

Código 3.44: Acción `extraerMensaje`

```
1 extraerMensaje :: FilePath -> IO ()
2 extraerMensaje st= do
3   byteString1 <- BS.readFile st
4   let (Right dynamicImage1) = decodeGif byteString1
5   let (ImageRGB8 imagen1) = dynamicImage1
6   let (Image w h pxs) = imagen1
7   let xs = listaCarac(menint (extraerLSB pxs))
8   putStrLn ("Mensaje: "++show( xs))
```

Código 3.44: Acción `extraerMensaje`

Esta acción realiza lo siguiente:

- Desde la línea 3 hasta la línea 6, se siguen los mismos pasos que se realizan en la acción `alglsbreplacement` (código 3.11) para obtener el ancho (`w`), la altura (`h`), y el vector con las codificaciones de los colores (`pxs`) de la imagen stego.
- En la línea 7, se obtiene la lista de caracteres del mensaje. Primero, se obtiene la lista con los dígitos asociados a los caracteres del mensaje evaluando la función `menint` (código 3.42) sobre la lista de los bits menos significativos de `pxs`, obtenidos mediante la función `extraerLSB` (código 3.39). Después, se consigue la lista de caracteres que forma el mensaje con la función `listaCarac` (código 3.43) y se asigna a la variable `xs`.
- En la línea 8 se utiliza la función `putStrLn` para imprimir en pantalla "Mensaje:", seguido del mensaje que se ha obtenido en el paso anterior.

### 3.4.2. Algoritmo Matrix Embedding

#### Librerías

Las librerías utilizadas para este algoritmo son:

Código 3.45: `Librerías para la extracción`

```
1 import qualified Data.Vector as V
2 import qualified Data.Vector.Storable as VS
3 import Codec.Picture.Gif
4 import Codec.Picture.Types
5 import qualified Data.ByteString as BS (readFile)
6 import qualified Data.Matrix as M
7 import qualified Data.List as L
8 import Data.Char
```

Código 3.45: `Librerías para la extracción`

- Las librerías `Data.Vector`, `Data.Vector.Storable`, `Codec.Picture.Gif`, `Codec.Picture.Types` y `Data.ByteString` se utilizan de la misma manera que en la extracción de los mensajes escondidos con el algoritmo LSB-replacement o LSB-matching (código 3.38).
- La librería `Data.Matrix` para multiplicar matrices y obtener las suma de los bits menos significativos, que corresponden con los bits del mensaje.
- La librería `Data.List`, para obtener la matriz con los vectores no nulos a partir de las listas que forman dichos vectores.
- La librería `Data.Char` para mostrar los caracteres que se habían escondido en la imagen stego.

### Algoritmo para extraer el mensaje

El esquema que se va a seguir es el siguiente:

1. Se obtendrá un vector con los bits menos significativos de la codificación binaria de los colores de la imagen stego que se ha usado para esconder el mensaje. Después se dividirá dicho vector en bloques de longitud $2^n - 1$.
2. Se multiplicará la matriz que hemos usado en el algoritmo de Matrix Embedding por cada uno de los bloques del paso anterior.
3. Del resultado de la operación anterior, se extraerá la codificación binaria del mensaje de la que se obtendrán sus caracteres.
4. Se definirá una acción similar a la del código 3.44 para mostrar el mensaje que se ha escondido

Primero, se obtienen los bits menos significativos con la función `extraerBits` (código 3.21).

A continuación, se define una función que saca las sumas de los bits menos significativos para cada bloque. Dichas sumas cumplen las ecuaciones de la sección 3.3.3, pero no se utiliza la suma binaria:

Código 3.46: Función `sacarSumaBits`

```
1 sacarSumaBits :: (Eq a, Num a) => V.Vector a -> Int -> [a]
2 sacarSumaBits pxs n =
3   let m= M.transpose (M.fromLists(vectoresNoNulos n))
4       bloques = dividirbloques pxs (2^n-1)
5       q= length bloques
6   in  concat [M.toList (m * (M.colVector (bloques Prelude.!!i))) | i<-[0..(q-1)]]
```

Código 3.46: Función `sacarSumaBits`

Obsérvese que `pxs` es el vector con la codificación de los colores de la imagen stego en sistema decimal, por lo que se utiliza la función `vecbin` (código 3.6) para obtener las codificaciones en binario. Después, se divide el vector en bloques de longitud $2^n - 1$ que se han utilizado para esconder los bloques del mensaje de n bits. Además, se utiliza la función `vectoresNoNulos` (código 3.22) para obtener la matriz con la que se ha codificado

el mensaje. Después, se concatenan los resultados de cada operación para formar una única lista.

Ahora, se le aplica módulo 2 a los elementos de la lista obtenida con la función anterior para obtener una lista con los bits del mensaje:

Código 3.47: Función `sacarBits`

```
1 sacarBits :: Integral b => [b] -> [b]
2 sacarBits xs =
3   L.map (`mod`2) xs
```

Código 3.47: Función `sacarBits`

Utilizando las funciones `menint` (código 3.42) y `listaCarac` (código 3.43), se obtienen los caracteres del mensaje.

Finalmente, se define la acción que recibe la ruta de la imagen stego y el tamaño de los bloques en los que se ha divido el mensaje al esconderlo, y muestra dicho mensaje:

Código 3.48: Acción `extraerMensajeMatrix`

```
1 extraerMensajeMatrix :: FilePath -> Int -> IO ()
2 extraerMensajeMatrix im n = do
3   byteString1 <- BS.readFile im
4   let (Right dynamicImage1) = decodeGif byteString1
5   let (ImageRGB8 imagen1) = dynamicImage1
6   let (Image w h pxs) = imagen1
7   let xs = listaCarac (menint(sacarBits (sacarSumaBits (extraerBits(vecbin(pxs)))
        n )))
8   putStrLn ("Mensaje: "Prelude.++show(xs))
```

Código 3.48: Acción `extraerMensajeMatrix`

Si se compara con el código 3.44, que extrae mensajes escondidos por el algoritmo LSB-replacement o LSB-matching, la única diferencia se encuentra en la línea 7, que obtiene la lista de caracteres del mensaje.

# CAPÍTULO 4

## Interfaz y manual de uso

En este capítulo, se describirá la implementación de una aplicación que permita esconder un mensaje en una imagen o extraer el mensaje de una imagen stego utilizando los algoritmos descritos en el capítulo anterior. Para construir dicha aplicación, se utilizará la librería gtk3. La interfaz de usuario consta de tres ventanas:

- Ventana de inicio: Que incluye dos botones para acceder a las otras ventanas.
- Ventana para esconder un mensaje: Que permite al usuario generar una imagen stego aplicando uno de los algoritmos implementados para esconder el mensaje.
- Ventana para extraer un mensaje: Permite al usuario escoger una imagen stego y obtener el mensaje oculto.

### 4.1– Librerías

Para diseñar la aplicación, se han importado las siguientes librerías:

Código 4.1: `Librerías para la aplicación`

```
1  import Control.Monad
2  import Control.Monad.IO.Class
3  import Graphics.UI.Gtk
4  import System.Glib.UTFString
5  import Data.Maybe
6  import Data.Char
7  import qualified Data.Vector as V
8  import qualified Data.Vector.Storable as VS
9  import Codec.Picture.Gif
10 import Codec.Picture.Saving
11 import Codec.Picture.Types
12 import qualified Data.ByteString as BS (readFile)
13 import qualified Data.ByteString.Lazy as BSL (writeFile)
14 import System.Random
15 import Data.Matrix as M
16 import Data.List as L
```

Código 4.1: `Librerías para la aplicación`

- La librería `Control Monad`, que contiene una función que ignora el resultado de la evaluación de una acción.
- La librería `Control.Monad.IO.Class`, que se utilizará para evitar que el programa siga funcionando una vez se cierre.
- La librería `Graphics.UI.Gtk` para construir la interfaz.
- Una librería que permita acceder a la cadena que forma el mensaje escrito en la interfaz. Esta librería es `System.Glib.UFTString`.
- La librería `Data.Maybe` para obtener un archivo.
- El resto de librerías son las librerías necesarias para implementar los algoritmos descritos en el capítulo anterior. Dichas librerías están descritas en las secciones 3.1.1, 3.2.1 y 3.3.1.

## 4.2– La ventana de inicio

### 4.2.1. Botones

La acción `mkButtonEmbedding` genera un botón. Su código es el siguiente:

Código 4.2: Acción `mkButtonEmbedding`

```
1 mkButtonEmbedding :: IO Button
2 mkButtonEmbedding = do
3   btn<-buttonNew
4   set btn [ buttonLabel := "Esconder un mensaje" ]
5   btn `on` buttonActivated $ do
6      main1
7   return btn
```

Código 4.2: Acción `mkButtonEmbedding`

En primer lugar, se crea un nuevo botón que será almacenado en la variable `btn`. Con `buttonLabel` se establece la etiqueta del botón, que en este caso es "Esconder un mensaje". Cuando se pulsa el botón, se ejecuta la acción `main1`, que genera la interfaz para insertar un mensaje escondido en una imagen.

Similarmente, se define la acción `mkButtonExtracting`, que crea un botón con la etiqueta "Extraer un mensaje de una imagen stego" y que al pulsarlo ejecuta la acción `main2`, que genera la interfaz que permite al usuario obtener el mensaje escondido por uno de los algoritmos esteganográficos descritos en el capítulo anterior:

Código 4.3: Acción `mkButtonExtracting`

```
1 mkButtonExtracting :: IO Button
2 mkButtonExtracting = do
3   btn<-buttonNew
4   set btn [ buttonLabel := "Extraer un mensaje de una imagen stego" ]
5   btn `on` buttonActivated $ do
6     main2
7   return btn
```

Código 4.3: Acción `mkButtonExtracting`

### 4.2.2. La interfaz

Ahora, se define una acción que genera una ventana que incluye a los botones definidos en la sección anterior:

Código 4.4: `main`

```
1  main :: IO ()
2  main = do
3    void initGUI
4    window <- windowNew
5    set window [ windowTitle      := "Aplicación esteganografía"
6               , windowResizable   := True
7               , windowDefaultWidth := 100
8               , windowDefaultHeight := 200 ]
9    grid <- gridNew
10   gridSetRowHomogeneous grid True
11
12   let attach x y w h item = gridAttach grid item x y w h
13
14   mkButtonEmbedding >>= attach 0 5 1 1
15   mkButtonExtracting >>= attach 1 5 1 1
16
17
18   containerAdd window grid
19
20   window `on` deleteEvent $ do
21     liftIO mainQuit
22     return False
23
24   widgetShowAll window
25
26   mainGUI
```

Código 4.4: `main`

- En la línea 3 se hace una llamada a `initGUI`. Esto siempre se hace para cualquier programa que use GTK+, ya que prepara a GTK+ para que funcione. Se utiliza `void` para que no nos devuelva ningún resultado.
- En la línea 4 se utiliza `windowNew` para generar una ventana opaca. A esta ventana se le pone de título "Aplicación esteganografía" y se da la opción de que se pueda cambiar el tamaño.
- En las líneas 9 y 10 se crea una cuadrícula con el comando `gridNew` y se establece que las filas se distribuyan homogéneamente.
- En la línea 12 se utiliza la asignación `let ...=...` para establecer la notación con la que se va a añadir un elemento a la cuadrícula. En este caso, las variables `x` e `y` denotan la posición donde se va a añadir el elemento `item`, y las variables `w` y `h` indican la anchura y la altura respectivamente de dicho elemento.
- En las líneas 14 y 15 se añaden los botones.

- Con el comando `containerAdd` en la línea 18, se añade la cuadrícula a la ventana
- El código escrito entre las líneas 20 y 22 permite que el programa deje de ejecutarse cuando se cierra la ventana.
- En la línea 24, se utiliza `widgetShowAll` para mostrar la ventana
- `mainGui` hace referencia al bucle principal que detecta movimientos como los del ratón o si se pulsa un botón.

La ventana que se genera es la siguiente:

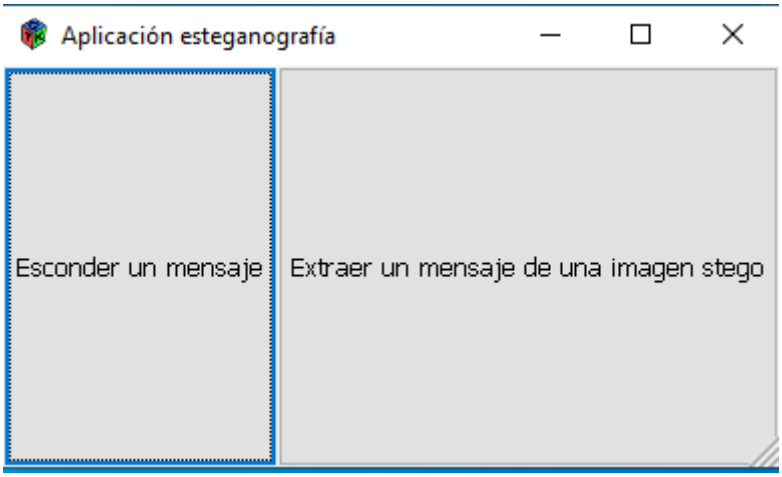


Figura 4.1: Ventana de inicio

Al pinchar el botón de la izquierda, se abre una ventana donde se puede esconder un mensaje mediante uno de los algoritmos implementados en una imagen GIF. Si se pulsa el botón de la derecha, se accede a una ventana en la que se puede escoger una imagen stego y obtener el mensaje escondido.

## 4.3– Ventana para esconder un mensaje

### 4.3.1. Seleccionar un fichero y escribir un mensaje

Primero, se define una acción que obtenga la ruta de un archivo:

Código 4.5: Acción `seleccionarFichero`

```
1 seleccionarFichero :: FileChooserClass self => self -> IO [Char]
2 seleccionarFichero fch = do
3  fichero <- fileChooserGetFilename fch
4  return (fromJust fichero)
```

Código 4.5: Acción `seleccionarFichero`

Como la acción `fileChooserGetFileName` devuelve un dato del tipo `IO (Maybe [Char])`, se utiliza `fromJust` para que sólo devuelva `IO[Char]`.

Ahora, para obtener el mensaje de una entrada que se añadirá a la ventana, se utiliza el siguiente código, que devuelve el texto escrito en el widget de entrada denotado como `display`:

Código 4.6: Acción `obtener`

```
obtener :: (EntryClass self, System.Glib.UTFString.GlibString b) =>self -> IO b
obtener display = do
   texto <- entryGetText display
   return texto
```

Código 4.6: Acción `obtener`

### 4.3.2. Botones

Falta crear los botones que permiten aplicar los algoritmos del capítulo anterior. En primer lugar, se tiene que el código para generar un botón que permita aplicar el algoritmo LSB-replacement es:

Código 4.7: Acción `mkButton1`

```
mkButton1 :: (FileChooserClass self1, FileChooserClass self2,EntryClass self3) =>
    self1 -> self3 -> self2 -> IO Button
mkButton1 im display st = do
  btn <- buttonNew
  set btn [ buttonLabel := "lsb-replacement" ]
  btn `on` buttonActivated $ do
    fichero <- seleccionarFichero im
    stego<- seleccionarFichero st
    texto<- obtener display
    alglsbreplacement fichero texto stego
  return btn
```

Código 4.7: Acción `mkButton1`

Este botón recibe como argumentos dos ficheros elegidos con el widget `fileChooser` y la entrada `display`, que se definirá más tarde a la hora de generar la ventana. El botón, al ser pinchado, recoge las rutas de los archivos `im` (la imagen cover) y `st` (la imagen stego) mediante la acción `seleccionarFichero` (código 4.5) y las almacena en las varables `fichero` y `stego` respectivamente. Además, almacena el texto que se recoge del widget `display` en la variable `texto`. Después, utiliza estos datos para aplicar el algoritmo LSB-replacement mediante la acción `alglsbreplacement` (código 3.11).

Similarmente, se crean los botones `mkButton2` y `mkButton3` que aplican los algoritmos LSB-matching, utilizando la acción `algLsbMatching` (código 3.19), y Matrix Embedding, utilizando `algMatrixEmbedding` (código 3.37):

Código 4.8: Acción `mkButton2`

```
mkButton2 :: (FileChooserClass self1, FileChooserClass self2,EntryClass self3) =>
    self1 -> self3 -> self2 -> IO Button
mkButton2 im display st = do
  btn <- buttonNew
  set btn [ buttonLabel := "lsb-matching" ]
  btn `on` buttonActivated $ do
    fichero <- seleccionarFichero im
    stego<- seleccionarFichero st
    texto<- obtener display
    algLsbMatching fichero texto stego
```

```
10  return btn
```

Código 4.8: Acción `mkButton2`

Código 4.9: Acción `mkButton3`

```
1  mkButton3 :: (FileChooserClass self1, FileChooserClass self2,EntryClass self3) =>
       self1 -> self3 -> self2 -> IO Button
2  mkButton3 im display st = do
3    btn <- buttonNew
4    set btn [ buttonLabel := "matrix embedding" ]
5    btn `on` buttonActivated $ do
6      fichero <- seleccionarFichero im
7      stego<- seleccionarFichero st
8      texto<- obtener display
9      algMatrixEmbedding fichero texto stego
10   return btn
```

Código 4.9: Acción `mkButton3`

### 4.3.3. La interfaz

La ventana que se genera con la acción `main1` tiene el siguiente código:

Código 4.10: `main1`

```
1  main1 :: IO ()
2  main1 = do
3    void initGUI
4    window <- windowNew
5    set window [ windowTitle      := "Esconder mensaje"
6               , windowResizable   := True
7               , windowDefaultWidth := 500
8               , windowDefaultHeight := 500 ]
9    display <- entryNew
10   set display [ entryEditable :=True
11               , entryXalign  := 0
12               , entryText    := "Escriba el texto que quiera esconder" ]
13
14   fch <- fileChooserWidgetNew FileChooserActionOpen
15   st  <-  fileChooserWidgetNew FileChooserActionSave
16
17   titfch <- labelNew (Just "Escoja la imagen que quiera utilizar. A continuación,
         indique el nombre de la imagen stego y dónde quiere guardarla")
18   titalg<- labelNew (Just "Escoja el algoritmo que quiera utilizar")
19
20   grid <- gridNew
21   gridSetRowHomogeneous grid True
22
23   let attach x y w h item = gridAttach grid item x y w h
24
25   attach 0 1 5 1 display
26   attach 0 2 5 1 titfch
27   attach 0 3 5 2 fch
```

```
28  attach 0 5 5 2 st
29  attach 0 7 5 1 titalg
30  mkButton1 fch display st >>= attach 0 8 1 1
31  mkButton2 fch display st >>= attach 1 8 1 1
32  mkButton3 fch display st >>= attach 2 8 1 1
33  containerAdd window grid
34
35  window `on` deleteEvent $ do
36    liftIO mainQuit
37    return False
38
39  widgetShowAll window
40
41  mainGUI
```

Código 4.10: `main1`

Como se puede observar, se han añadido más elementos con respecto al código 4.4:

- Se ha añadido una entrada en la línea 9, donde se va a escribir el mensaje que se va a ocultar. La entrada se puede editar y el texto que aparece por defecto es "Escriba el texto que quiera esconder".
- En la línea 14, se añade un widget que nos permite abrir un archivo.
- En la línea 15 también se añade un widget de tipo `fileChooser`, pero esta vez, la acción que realiza es la de guardar un archivo.
- En las líneas 17 y 18, se añaden etiquetas para que el usuario sepa los pasos a seguir para esconder el mensaje deseado.
- Los widgets, las etiquetas y los botones definidos en esta sección se añaden a la cuadrícula igual que se hizo en el código 4.4

Una vez pulsado el botón de la izquierda en la ventana de inicio (4.1), se abre la ventana generada por el código anterior y el usuario puede interactuar con ella de la siguiente manera:

- En la barra del principio, el usuario puede escribir. La idea es escribir en esta barra el mensaje que se quiera esconder.
- En el selector de archivos, el usuario puede acceder a los ficheros de las distintas carpetas que se encuentran en el equipo. La ruta del fichero seleccionado se recogerá y se usará como dato para los algoritmos. El fichero seleccionado corresponde a la imagen cover.
- En el segundo selector de archivos, el usuario puede poner nombre a un nuevo fichero y seleccionar en qué carpeta quiere guardarlo. Esta información es recogida para generar la imagen stego.
- Pulsando alguno de los 3 botones de abajo, se aplica el algoritmo correspondiente para esconder el mensaje escrito arriba.

La ventana generada quedaría de la siguiente manera:

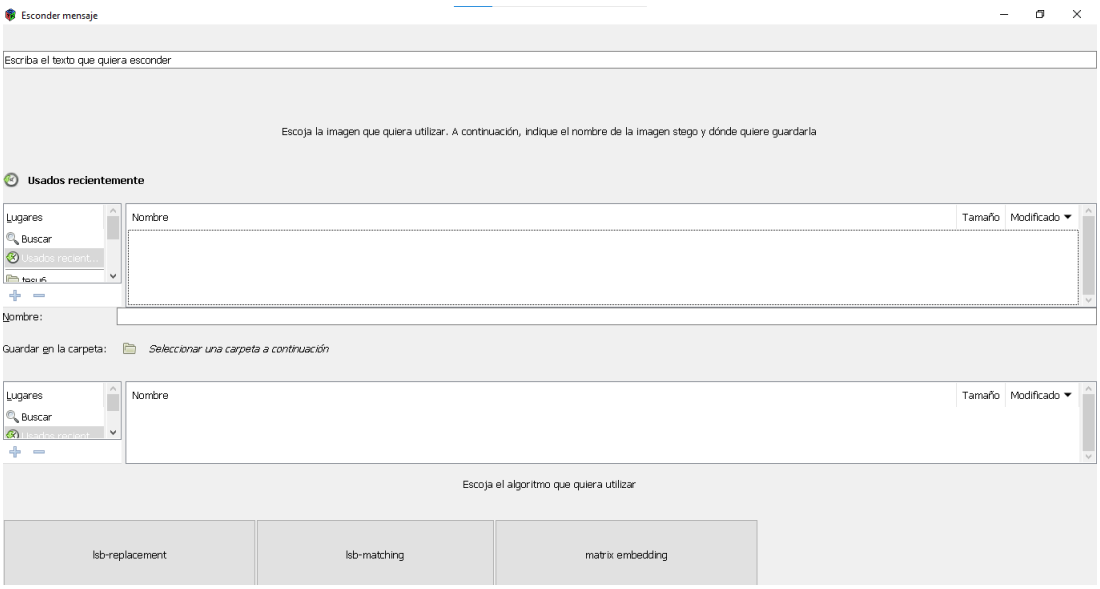


Figura 4.2: Ventana para esconder un mensaje

## 4.4– Ventana para extraer un mensaje

### 4.4.1. Botones

En esta sección, lo que se pretende es desarrollar una interfaz que permita al usuario extraer el mensaje escondido en una imagen stego, conociendo el algoritmo con el que se ha escondido dicho mensaje. Para lograr esto se definirán botones que, al pulsarlos, apliquen los algoritmos de extracción definidos en el capítulo anterior. Sin embargo, en esta ocasión, es necesario que el algoritmo no imprima en pantalla el mensaje, sino que se pretende que aparezca en la ventana. Es por esto que se modifican los códigos 3.44 y 3.48 y, en vez de utilizar el comando `putStrLn` en la última línea, se utiliza el comando `return`:

Código 4.11: Acción `extraerMensaje2`

```
1 extraerMensaje2 :: FilePath -> IO [Char]
2 extraerMensaje2 st= do
3   byteString1 <- BS.readFile st
4   let (Right dynamicImage1) = decodeGif byteString1
5   let (ImageRGB8 imagen1) = dynamicImage1
6   let (Image w h pxs) = imagen1
7   let xs = listaCarac(menint (extraerLSB pxs))
8   return xs
```

Código 4.11: Acción `extraerMensaje2`

Código 4.12: Acción `extraerMensajeMatrix2`

```
extraerMensajeMatrix2 :: FilePath -> Int -> IO [Char]
extraerMensajeMatrix2 im n = do
  byteString1 <- BS.readFile im
  let (Right dynamicImage1) = decodeGif byteString1
  let (ImageRGB8 imagen1) = dynamicImage1
  let (Image w h pxs) = imagen1
  let xs = listaCarac (menint(sacarBits (sacarSumaBits (extraerBits(vecbin(pxs)))
       n )))
  return xs
```

Código 4.12: Acción `extraerMensajeMatrix2`

Ahora, se pueden definir los botones que permitan al usuario extraer el mensaje:

Código 4.13: Acción `mkButtonL`

```
mkButtonL :: (FileChooserClass self, EntryClass o) => self -> o -> IO Button
mkButtonL im display = do
  btn<-buttonNew
  set btn [ buttonLabel := "lsb-replacement ólsb-matching" ]
  btn `on` buttonActivated $ do
     fichero <- seleccionarFichero im
     xs<- extraerMensaje2 fichero
     set  display [ entryText := xs ]
  return btn
```

Código 4.13: Acción `mkButtonL`

El botón generado con el código anterior extrae un mensaje que ha sido escondido mediante el algoritmo LSB-replacement o el algoritmo LSB-matching. En primer lugar, se obtiene la ruta del fichero con la imagen stego con la acción `seleccionarFichero` (código 4.5) y extrae el mensaje con la acción definida anteriormente (código 4.11) y se almacena en la variable `xs`. Después, se modifica la entrada a la que se le denominará `display` para que muestre dicho mensaje.

Similarmente, se definen los botones para extraer un mensaje escondido con el algoritmo Matrix Embedding. En este caso, se crean 3 botones: uno para cuando el mensaje se ha dividido en bloques de 2 bits a la hora de esconderlo (`mkButtonM2`), otro para mensajes que se dividieron en bloques de 3 bits (`mkButtonM3`), y otro para bloques de 4 bits (`mkButtonM4`).

Código 4.14: Acción `mkButtonM2`

```
mkButtonM2 :: (FileChooserClass self, EntryClass o) => self -> o -> IO Button
mkButtonM2 im display = do
  btn<-buttonNew
  set btn [ buttonLabel := "Matrix Embedding con bloques de 2 bits" ]
  btn `on` buttonActivated $ do
     fichero <- seleccionarFichero im
     xs<- extraerMensajeMatrix2 fichero 2
     set  display [ entryText := xs ]
  return btn
```

Código 4.14: Acción `mkButtonM2`

Código 4.15: Acción `mkButtonM3`

```
mkButtonM3 :: (FileChooserClass self, EntryClass o) => self -> o -> IO Button
mkButtonM3 im display = do
  btn<-buttonNew
  set btn [ buttonLabel := "Matrix Embedding con bloques de 3 bits" ]
  btn `on` buttonActivated $ do
     fichero <- seleccionarFichero im
     xs<- extraerMensajeMatrix2 fichero 3
     set  display [ entryText := xs ]
  return btn
```

Código 4.15: Acción `mkButtonM3`

Código 4.16: Acción `mkButtonM4`

```
mkButtonM4 :: (FileChooserClass self, EntryClass o) => self -> o -> IO Button
mkButtonM4 im display= do
  btn<-buttonNew
  set btn [ buttonLabel := "Matrix Embedding con bloques de 4 bits" ]
  btn `on` buttonActivated $ do
     fichero <- seleccionarFichero im
     xs<-extraerMensajeMatrix2 fichero 4
     set  display [ entryText := xs ]
  return btn
```

Código 4.16: Acción `mkButtonM4`

### 4.4.2. La interfaz

Se define la función `main2`:

Código 4.17: `main2`

```
main2 :: IO ()
main2 = do
  void initGUI
  window <- windowNew
  set window [ windowTitle      := "Extraer mensaje"
             , windowResizable   := True
             , windowDefaultWidth := 500
             , windowDefaultHeight := 500 ]
  display <- entryNew
  set display [ entryEditable :=False
              , entryXalign  := 0
              , entryText    := "" ]
  fch <- fileChooserWidgetNew FileChooserActionOpen
  titdisplay<-labelNew (Just "Mensaje:")
  titfch<- labelNew (Just "Escoja la imagen stego")
  titalg<- labelNew (Just "Escoja el algoritmo que haya sido utilizado para
      guardar el mensaje")
  grid <- gridNew
  gridSetRowHomogeneous grid True

```

```
20  let attach x y w h item = gridAttach grid item x y w h
21  attach 0 6 5 1 titdisplay
22  attach 0 7 5 1 display
23  attach 0 1 5 1 titfch
24  attach 0 2 5 2 fch
25  attach 0 4 5 1 titalg
26  mkButtonL fch display >>= attach 0 5 1 1
27  mkButtonM2 fch display >>= attach 1 5 1 1
28  mkButtonM3 fch display >>= attach 2 5 1 1
29  mkButtonM4 fch display >>= attach 3 5 1 1
30
31  containerAdd window grid
32
33  window `on` deleteEvent $ do
34    liftIO mainQuit
35    return False
36
37  widgetShowAll window
38
39  mainGUI
```

Código 4.17: `main2`

Como se puede observar, la estructura de este código es similar al de la ventana para esconder un mensaje (código 4.10), diferenciándose en los siguientes puntos:

- En la línea 10, se establece que la entrada no sea editable, ya que se modificará con los botones creados para mostrar el mensaje extraído.
- No se utiliza ningún widget para guardar una imagen
- Las etiquetas para guiar al usuario son distintas
- Se añaden botones que inician los algoritmos de extracción

El usuario puede interactuar de la siguiente manera con la interfaz:

- Puede escoger una imagen que se encuentre en su equipo. Esta imagen será la imagen stego.
- Puede pulsar uno de los botones para extraer el mensaje escondido con el algoritmo que indica la etiqueta del botón.
- En la barra de abajo, el usuario puede visualizar el mensaje extraído.

La ventana que se abre al pulsar el botón de la derecha en la ventana 4.1 es la siguiente:


Figura 4.3: Ventana para extraer un mensaje

# CAPÍTULO 5

## Conclusiones

La esteganografía es una práctica que permite ocultar las comunicaciones entre distintas partes. Al igual que la criptografía, su objetivo es la de proteger la información de ojos ajenos a la comunicación, sin embargo, la manera en la que se oculta dicha información es distinta. Estos dos métodos no deberían considerarse excluyentes, sino que podría ser muy ventajoso considerarlos como complementarios. La esteganografía oculta el hecho de que se esté produciendo un intercambio de información y, en caso de ser descubierta, el hecho de que esté cifrado el mensaje supone otra barrera de protección. Asimismo, el intercambio de un mensaje cifrado, levanta muchas sospechas y será susceptible a ataques, por lo que si este mensaje no es visible, reduce las posibilidades de que sea descubierto.

Hay distintos tipos de esteganografía, pero la más popular es la esteganografía en imágenes. No sólo resulta un archivo en el que se puede esconder mayor cantidad de información en comparación con otros medios que se pueden usar en esteganografía, sino que resulta un archivo muy común en internet. No es extraño hacernos fotos constantemente y compartirlas en redes sociales, es por eso que no sería de extrañar que entre la cantidad de imágenes que vemos encontremos alguna en la que se le haya escondido algún mensaje.

Además de para poder comunicarse en secreto, la esteganografía puede utilizarse simplemente para guardar información. Algunas veces se recoge la información del autor de la imagen y se oculta utilizando algún algoritmo esteganográfico. De este modo, la imagen no sufre ninguna modificación perceptible para el ojo humano. Además, a diferencia de las marcas de agua visibles, esta información está oculta y pasará desapercibida para alguien que quiera apropiarse de la autoría de la imagen y no la borrará.

En la implementación de este trabajo, se han tratado algunos algoritmos estaganográficos básicos, siendo el LSB-replacement el más sencillo de implementar. Sin embargo, este algoritmo es fácil de detectar ya que, en el peor de los casos, es necesario cambiar tantos bits de la imagen como bits tenga la codificación binaria del mensaje y, además de esto, el hecho de que no exista la misma probabilidad de sumar 1 que restárselo a las codificaciones binarias de los colores de la imagen provoca anomalías estadísticas. Una posible solución sería esconder el mensaje en codificaciones aleatorias, dificultando la extracción del mensaje. Sin embargo, para mitigar las anomalías estadísticas, es recomendable usar el algoritmo LSB-matching, aunque el problema de que pueda ser necesario modificar una gran cantidad de bits persiste. Es por esto que el algoritmo Matrix Embedding es más adecuado si se quieren evitar estos problemas. En dicho algoritmo, la modificación de bits

es la mínima posible, ya que por cada bloque del mensaje que se esconde, sólo es necesario cambiar un bit en el peor de los casos. El principal problema de este algoritmo es que utiliza más bits a la hora de esconder un mensaje, por lo que necesita imágenes de mayor tamaño si se aplica con textos más largos. Una posible mejora es utilizar bloques de menor tamaño, aunque esto aumentaría el número de modificaciones necesarias en la imagen. Otra posible solución es intentar acortar el texto aumentando las abreviaturas. Por último, otro remedio sería utilizar varias imágenes para ocultar el mensaje, de manera que si una imagen stego se ve comprometida, el mensaje en dicha imagen estará incompleto. Teniendo en cuenta lo anterior, si comparamos el algoritmo Matrix Embedding con los algoritmos LSB-replacement y LSB-matching, el primero resulta más eficaz que los otros dos, debido a que puede implementarse tanto con imágenes GIF como imágenes JPEG, y que los mensajes escondidos con este algoritmo son más difíciles de detectar.

A la hora de implementar dichos algoritmos, ha sido necesario comprender cómo modifican los datos de la imagen los distintos algoritmos. Para ello, se han consultado las referencias citadas en el capítulo 2. Una vez entendido cómo funciona el algoritmo LSB-replacement [22] [26], el algoritmo LSB-matching [22] no es difícil de aprender, debido a la similitud entre ambos. El algoritmo Matrix Embedding suponía un reto distinto a la hora de insertar el mensaje: mientras que en los algoritmos anteriores se trabajaba con los bits menos significativos de uno en uno, en este caso se trabajaba con un conjunto de ellos. Además, se ha tenido que hacer un estudio sobre las modificaciones necesarias en cada caso teniendo en cuenta las relaciones que se han establecido. Para poder comprender mejor este algoritmo, se ha complementado la información proporcionada en [5] con la que aporta la referencia [32]. Esta comprensión de cómo funcionan los algoritmos implementados es lo que ha hecho posible poder proponer algoritmos de extracción para los mensajes escondidos con ellos. Sin embargo, lo que ha presentado mayor dificultad ha sido aprender cómo extraer la información de las imágenes en Haskell. Para ello, se han consultado ejemplos [19] y la documentación de la librería [16].

Por otro lado, también se ha diseñado una aplicación que permite al usuario aplicar estos algoritmos de inserción y de extracción de mensajes. Se ha estudiado cómo generar una interfaz utilizando como guía el ejemplo incluido en [20] para crear una nueva ventana, una entrada y los botones. También se ha consultado [15] para añadir un widget que permita seleccionar ficheros y etiquetas que guíen al usuario en la interfaz, y, además, buscar acciones que permitan recoger la información obtenida en el seleccionador de archivos y el texto escrito en la entrada.

# Bibliografía


[1] F.J. Martín Mateos *Documentación de un TfE en LaTeX* Personal edition, 2020.

[2] Zaid Al-Omari y Ahmad T. Al-Taani *A Survey on Digital Image Steganography* , ICIT, 2015

[3] R.Bala Krishnan, Prasanth Kumar Thandra y M.Sai Baba1 *An Overview of Text Steganography*, International Conference on Signal Processing, Communications and Networking,2017, Chennai, INDIA

[4] Fatiha Djebbar , Beghdad Ayad, Karim Abed Meraim y Habib Hamam *Comparative study of digital audio steganography techniques* , Djebbar et al. EURASIP Journal on Audio, Speech, and Music Processing, 2012

[5] M. Giménez Aguilar *Desarrollo y análisis de algoritmos de esteganografía* , 2017

[6] Haskell *Control.Monad*. Consultado en https://hackage.haskell.org/package/base-4.18.0.0/docs/Control-Monad.html

[7] Haskell *Control.Monad.IO.Class*. Consultado en https://hackage.haskell.org/package/base-4.18.0.0/docs/Control-Monad-IO-Class.html

[8] Haskell *Data.ByteString*. Consultado en https://hackage.haskell.org/package/bytestring-0.11.4.0/docs/Data-ByteString.html

[9] Haskell *Data.ByteString.Lazy*. Consultado en https://hackage.haskell.org/package/bytestring-0.11.4.0/docs/Data-ByteString-Lazy.html

[10] Haskell *Data.Char*. Consultado en https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-Char.html

[11] Haskell *Data.List*. Consultado en https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-List.html

[12] Haskell *Data.Matrix*. Consultado en https://hackage.haskell.org/package/matrix-0.3.6.1/docs/Data-Matrix.html

[13] Haskell *Data.Vector*. Consultado en https://hackage.haskell.org/package/vector-0.13.0.0/docs/Data-Vector.html

[14] Haskell *Data.Vector.Storable.* Consultado en https://hackage.haskell.org/package/vector-0.13.0.0/docs/Data-Vector-Storable.html

[15] Haskell *gtk3.* Consultado en gtk3: Binding to the Gtk+ 3 graphical user interface library (haskell.org)

[16] Haskell *Juicy Pixels.* Consultado en https://hackage.haskell.org/package/JuicyPixels

[17] Haskell *System.Glib.UTFString* . Consultado en https://hackage.haskell.org/package/glib-0.13.10.0/docs/System-Glib-UTFString.html

[18] Haskell *System.Random* . Consultado en https://hackage.haskell.org/package/random-1.2.1.1/docs/System-Random.html

[19] Mark Karpov *Image processing with Juicy Pixels and Repa* , 2016. Consultado en https://www.stackbuilders.com/blog/image-processing/

[20] Mark Karpov *Creating a GUI application in Haskell* , 2016. Consultado en Stack Builders - Creating a GUI application in Haskell

[21] Erick Lahura *La relación dinero-producto, brecha del producto e inflación subyacente: Una introducción a la teoría de Wavelets y sus aplicaciones* 2004

[22] Daniel Lerch Hostalot *Esteganografía LSB en imágenes y audio* , `https://daniellerch.me/stego/lab/intro/lsb-es/`

[23] Józef Lubacz, Wojciech Mazurczyk, y Krzysztof Szczypiorski *Principles and Overview of Network Steganography* , IEEE Communications Magazine, 2014

[24] Anandaprova Majumdera, y Suvamoy Changder *A Novel Approach for Text Steganography: Generating Text Summary using Reflection Symmetry*, International Conference on Computational Intelligence: Modeling Techniques and Applications (CIMTA), 2013

[25] Ms J Mary Jenifer, Dr S.Raja Ratna, Dr J. B. Shajilin Loret y Ms D.Merlin Gethsy *A survey on different Video Steganography Techniques* , Proceedings of the 2nd International Conference on Trends in Electronics and Informatics (ICOEI 2018)

[26] T. Morkel , J.H.P. Eloff y M.S. Olivier *An overview of image steganography*, Information and Computer Security Architecture (ICSA) Research Group,Department of Computer Science, University of Pretoria, 0002, Pretoria, South Africa

[27] R. J. Mstafa y K. M. Elleithy *A Novel Video Steganography Algorithm in the Wavlelet Domain Based on the KLT Tracking Algorithm and BCH Codes*, IE Conference on Systems, Applications and Technology pp 1-7, 2015

[28] M. Munikar *Image Steganography: Basic Concepts and Proposed Algorith*, 2016

[29] Noda, Hideki y Furuta, Tomofumi y Niimi, Michiharu y Kawaguchi, Eiji *Video steganography based on bit-plane decomposition of wavelet-transformed video*, Security, Steganography, and Watermarking of Multimedia Contents VI, 2004

[30] Simmons, G *The prisoners problem and the subliminal channel*, CRYPTO, 1983

[31] Derek Upham *Jsteg* , `https://zooid.org/~paul/crypto/jsteg`

[32] Westfeld, A. *F5—A Steganographic Algorithm. In: Moskowitz, I.S. (eds) Information Hiding* , IH 2001. Lecture Notes in Computer Science, vol 2137. Springer, Berlin, Heidelberg. `https://doi.org/10.1007/3-540-45496-9_21`