FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” - UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

KAUÊ DORETTO GRECCHI

O DESENVOLVIMENTO DE UM SISTEMA DE ESTEGANOGRAFIA

MARÍLIA
2008

KAUÊ DORETTO GRECCHI

O DESENVOLVIMENTO DE UM SISTEMA DE ESTEGANOGRAFIA

Monografia apresentada ao Curso de Ciência da Computação, do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador:

Prof. Dr. Antônio Carlos Sementille

MARÍLIA
2008

GRECCHI, Kauê Doretto
O desenvolvimento de um sistema de esteganografia / Kauê Doretto Grecchi; orientador: Antônio Carlos Sementille. Marília, SP: [s.n.], 2008. 113 f.

Monografia (Graduação em Ciência da Computação) – Centro Universitário Eurípides Soares da Rocha – Fundação de Ensino Eurípides Soares da Rocha.

1. Segurança. 2. Esteganografia 3. Ocultação de dados

CDD: 005.1068

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Kauê Doretto Grecchi

O DESENVOLVIMENTO DE UM SISTEMA DE ESTEGANOGRAFIA

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 10,0 (DEZ)

Orientador: Antonio Carlos Sementille

1°. Examinador: Ildeberto Aparecido Rodello

2°. Examinador: Valter Vieira de Camargo

Marília, 27 de junho de 2008.

Mais que somente este trabalho, dedico minha vida aos meus pais, que nunca mediram esforços para que eu me tornasse a pessoa que sou.

AGRADECIMENTOS

Aos meus pais, Marli e Edward, por terem feito o possível e o impossível para me criar e educar, e à minha irmã Karol pelos lanches da tarde.

Às minhas tias Maria Carolina e Ângela e meu tio João que me serviram como incentivo ao estudo científico, minha tia Marta por ajudar, em parte da minha educação e minha tia Márcia pelas conversas quando estava contrariado.

À minha tia Neusa pela hospitalidade e atenção enquanto eu me descabelava implementando as funções de compactação e maldizia o senhor Huffman.

À minha namorada Janaína pelo apoio nas horas difíceis e pela companhia de todos os dias.

Aos meus amigos mais próximos de faculdade: Mariana, Priscila, Fábio e Vanessa pela companhia e ajuda mútua entre tantos trabalhos e provas. Se não fossem vocês eu teria pirado neste último ano.

Aos meus amigos Gilmar, Júnior, Zeca, Barba e Arthur pelas situações cômicas, festas e finais de semana para descansar a cabeça (“esquece o exame e vamos tomar uma”), afinal “o importante é ter história pra contar”.

Aos colegas de trabalho da agência Lins e amigos de viagem: Ricardo, Guilherme, Márcia e Karina, pelo silêncio das idas e o “caldo de cana” das voltas.

Ao meu orientador Sementille, por ter me apresentado a técnica estudada neste trabalho quando pedi um tema diferente e novo, e por ter me ajudado de diversas maneiras na elaboração. Principalmente no dia que me disse “nós temos duas semanas pra entregar o trabalho pronto se quiser apresentar agora em Junho.”. Nada como um bom incentivo!

Aos professores Beto e Valter pelas aulas durante o curso e por terem aceito fazer parte da banca examinadora praticamente na última hora.

Acho que vírus de computador deveriam contar como vida. Acho que diz algo sobre a natureza humana que a única forma de vida que criamos seja até então puramente destrutiva. Criamos vida à nossa imagem.

(Stephen Hawking)

GRECCHI, Kauê Doretto. **O desenvolvimento de um sistema de esteganografia.** 2008. 113 f. Dissertação (Graduação em Ciência da Computação – Centro Universitário Eurípides Soares da Rocha – Fundação de Ensino Eurípides Soares da Rocha, Marília, 2008).

RESUMO

A popularização dos dispositivos de armazenamento e das redes de comunicação tornou necessária a proteção das informações que são armazenadas e transmitidas, do acesso não autorizado. Algumas técnicas de segurança digital tornam impossível o conhecimento do significado das informações, enquanto outras ocultam sua própria existência. A esteganografia é uma técnica de ocultação de dados que torna possível a inserção de informações secretas, sem que haja mudanças perceptíveis, em um objeto recipiente aparentemente inócuo. A partir do estudo feito neste trabalho, escolheu-se as técnicas consideradas mais relevantes para o projeto e implementação de um sistema de esteganografia em arquivos de imagem no formato bitmap aliada à compactação de dados.

Palavras-chave: Segurança. Esteganografia. Ocultação de dados.

GRECCHI, Kauê Doretto. **O desenvolvimento de um sistema de esteganografia.** 2008. 113 f. Dissertação (Graduação em Ciência da Computação – Centro Universitário Eurípides Soares da Rocha – Fundação de Ensino Eurípides Soares da Rocha, Marília, 2008).

ABSTRACT

The popularization of storage devices and communication networks created the need to protect the stored and transmitted information from non-authorized access. Some techniques of digital security make the acknowledgement of the information impossible, while others hide it´s very existence. Steganography is a data hiding technique that makes possible the insertion of secret information, without leaving major evidences, in an apparently innocuous object. From the survey made in this paper, the most relevant techniques have been chosen to the project and implementation of a bitmap format image file steganography system allied to data compression.

Keywords: Security. Steganography. Data hiding.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE GRÁFICOS

LISTA DE EQUAÇÕES

LISTA DE ABREVIATURAS

AHA: Árvore de Huffman Adaptativo

ASCII: American Standard Code for Information Interchange (Código Padrão Americano Para Intercâmbio de Informação)

BMP: Bitmap

DCT: Discrete Cosine Transform (Transformada Discreta de Coseno)

DEA: Drug Enforcement Administration (Administração de Repressão às Drogas)

DFT: Discrete Fourier Transform (Transformada Discreta de Fourier)

DICOM: Digital Imaging and Communication in Medicine (Imagem e Comunicação Digital em Medicina)

EOF: End of File (Fim de Arquivo)

JPEG: Joint Photograph Experts Group (Grupo de Especialistas em Fotografia em Conjunto)

LSB: Least Significant Bit (Bit Menos Significativo)

LZW: Algoritmo de compactação Lempel-Ziv-Welch

NYT: Not-Yet-Transmitted (Ainda-Não-Transmitido)

SUMÁRIO

INTRODUÇÃO

A imensa quantidade de informações e conhecimento produzidos desde o final do século XX, mudou radicalmente as relações mundiais. A política, a sociedade, a economia, a indústria, e até o crime, tiveram que criar novos conceitos, e reavaliar os antigos, para se adaptarem às novas exigências do mundo. A rede mundial de computadores (Internet) está entre as criações mais importantes que contribuíram para a dispersão de conhecimento, transpondo fronteiras entre países e se somando à globalização. Devida à característica de não ser um lugar físico, mas virtual, a Internet em seu início era um domínio livre, possibilitando a qualquer pessoa consultar as informações nela publicadas ou adicionar sua própria contribuição, praticamente sem restrições. Com o aumento cada vez maior do número de usuários e da quantidade de informação disponível, as informações que lá trafegavam passaram, de simples compartilhamento pessoal e acadêmico, a bens com alto valor intelectual, principalmente nas áreas militar, comercial e industrial. As pessoas e instituições encontraram um meio muito eficiente e barato de comunicação, tanto para fins benéficos, como publicação de material acadêmico, quanto maléficos, como roubo e destruição de informações, com ou sem fins financeiros.

Diante das novas ameaças no ambiente virtual, foi necessária a criação de técnicas para restringir o acesso indevido e garantir que informações confidenciais sejam transmitidas em segurança até seu destino. Várias foram desenvolvidas e aplicadas, umas com sucesso e são utilizadas até hoje, outras tiveram suas falhas expostas e exploradas e foram abandonadas. Uma técnica de ocultação de dados criada recentemente e baseada em uma técnica muito antiga é a esteganografia.

Na esteganografia clássica, uma mensagem que se deseja transmitir secretamente é inserida em um objeto que não levante suspeitas sobre a existência da mesma. Baseado neste método foi criada a esteganografia digital, que consiste na ocultação de dados em um arquivo de cobertura, sem alterar as características de ambos.

Considerando o contexto exposto, este trabalho apresenta um estudo sobre a esteganografia e o desenvolvimento de um software que implementa a técnica aliada, à compactação de dados, para que seja melhor aproveitado o espaço disponível no arquivo de cobertura. Após a implementação foram realizados também testes e análises para medir o desempenho do sistema.

O trabalho possui a seguinte organização:

No Capítulo 1 é feito o levantamento das características da esteganografia, desde suas origens até as técnicas mais recentes.

No Capítulo 2 são reunidos e explicados os principais métodos de implementação de sistemas de esteganografia e suas classificações.

No Capítulo 3 é apresentado o estudo feito sobre técnicas de compactação de dados e o algoritmo de compactação de Huffman.

No Capítulo 4 é detalhado o desenvolvimento do sistema proposto neste trabalho.

No Capítulo 5 são aplicados e analisados testes de eficiência do software sob vários parâmetros diferentes.

O Capítulo 6 conclui o trabalho mostrando a relação entre os objetivos propostos e os obtidos e propõe trabalhos futuros.

CAPÍTULO 1 - ESTEGANOGRAFIA: UMA VISÃO GERAL

A esteganografia é uma técnica de segurança de informação que permite que dados sejam transmitidos de modo que sua própria existência não possa ser notada. Este Capítulo é resultado do estudo da técnica e seus conceitos, sobre os quais o trabalho é estruturado.

1.1 Definição

A palavra esteganografia tem origem grega e significa “escrita encoberta” (ANDERSON et al, 1999), podendo também ser definida como a arte da ocultação de mensagens. A técnica é baseada no uso de um objeto de cobertura, que irá receber a mensagem que se deseja transmitir. O objeto de cobertura pode ser um objeto comum qualquer ou mesmo uma outra mensagem, sendo possível extrair dela a mensagem original, sabendo-se o método de extração. Não deve haver nenhuma alteração física no objeto de cobertura que evidencie a presença do conteúdo embutido.

A ocultação da existência de uma mensagem é necessária quando se deseja transmiti-la em segurança por meios hostis. Um meio hostil é aquele em que agentes externos têm a intenção de extrair, alterar ou destruir a mensagem oculta, com a intenção de conhecer o segredo ou evitar sua recepção. Em sua forma clássica é a ocultação de uma mensagem em uma mensagem de cobertura, de modo que as características da segunda não sejam visivelmente alteradas (ANDERSON; PETITCOLAS, 1998). Na forma digital dados são inseridas em outros tipos de dados como arquivos de imagem, vídeo e som, entre outros.

1.2 Terminologia

Com a criação e o crescente estudo da esteganografia, muitas formas de descrever suas características e componentes foram definidos, embora os autores de trabalhos muitas vezes divergiam nos termos utilizados.

Para que houvesse um consentimento foi realizado em Cambridge, Inglaterra, o *First Information Hiding Workshop*, em 1996, onde Pfitzmann (1996) propôs que a nomenclatura dos termos relacionados a esteganografia e *fingerprinting* (marcação para reconhecimento da

origem, direitos autorais e número de série) deveria seguir um padrão de sufixos e prefixos, dependendo da aplicação.

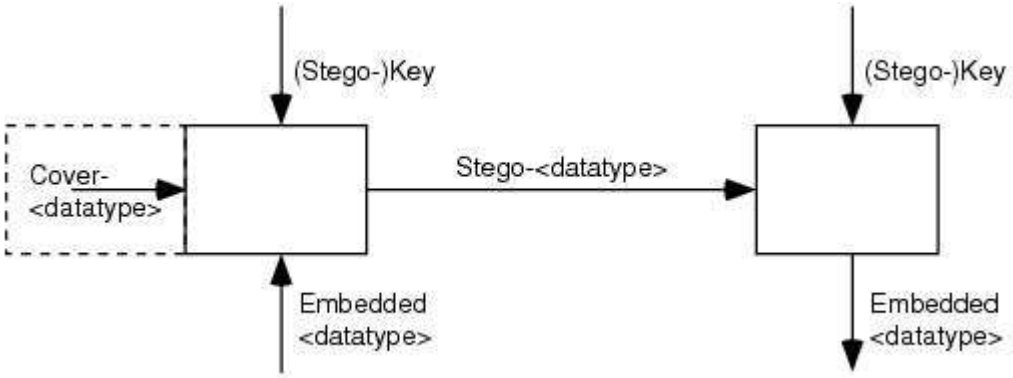


Figura 1 - Nomenclatura dos termos usados em esteganografia (Fonte: PFITZMANN, 1996).

Seguindo a proposta do autor, neste trabalho será utilizada a seguinte nomenclatura, ilustrada pela Figura 1:

<tipo de dado> de cobertura (*cover-<datatype>*): Nomenclatura dada a um objeto que irá receber os dados a serem escondidos. Pode-se substituir o sufixo “cobertura” por recipiente”. Exemplo: imagem de cobertura, arquivo de cobertura, objeto de cobertura, vídeo de cobertura.

<tipo de dado> embutido (*embedded <datatype>*): Dados que estejam esteganografados em um objeto de cobertura. Também pode-se substituir a palavra “embutido” por “oculto” ou “escondido”. Exemplos: dado embutido, mensagem embutida, imagem embutida.

estego-<tipo de dado> (*Stego-<datatype>*): Arquivo que contenha conteúdo esteganografado dentro de si. Também chamado de estego-objeto. Exemplos: estego-imagem, estego-áudio, estego-arquivo.

(estego-)chave (*(stego-)key*): Chave utilizada para proteger os dados embutidos e permitir ou negar a extração dos dados.

1.3 Utilização

A garantia de que a mensagem oculta está bem protegida pela mídia recipiente torna a esteganografia uma técnica ideal para armazenamento e transmissão de informações importantes. Da mesma maneira que a esteganografia ajuda a manter dados em segurança, pode também auxiliar seu furto, permitindo a um espião colocar tais dados em imagens que não levantariam suspeita em um sistema de vigilância de tráfego de rede ou em uma revista de segurança.

A esteganografia não serve somente para ocultação de mensagens. Segundo Anderson e Petitcolas (1998), marcas d'água digitais invisíveis podem ser inseridas em imagens, o que auxiliaria na identificação de alterações. Através desta mesma técnica é possível marcar uma imagem que esteja sob direitos autorais. Mais aplicações podem ser o *fingerprinting*, a monitoração de propagandas em rádio, detecção de traidores e inserção dos dados de um paciente em imagens médicas DICOM (*Digital Imaging and Communication in Medicine*). As imagens DICOM carregam consigo os dados do paciente, a data e o nome do médico. Contudo, algumas vezes a ligação entre os dados e a imagem é perdida, portanto a inserção do nome do paciente na imagem ajudaria a manter a consistência das informações (ANDERSON; PETITCOLAS, 1998).

Há também o temor de que terroristas utilizem a técnica para comunicação e planejamento de ataques. Em 2001 o jornalista Jack Kelley publicou, em sua coluna de tecnologia no jornal *The USA Today*, que Osama bin Laden teria planejado os ataques terroristas, em 11 de setembro daquele ano, ao *World Trade Center*, utilizando a esteganografia como forma de comunicação durante o planejamento (KELLEY, 2001). Com a descoberta da invenção dessa e de várias outras reportagens (BANVILLE, 2004) Kelley foi demitido do jornal.

Em 2007 o traficante colombiano Juan Carlos Ramirez-Abadia, um dos mais procurados do mundo, foi preso no Brasil acusado, entre outros crimes, de lavagem de dinheiro do tráfico de drogas (TERRA, 2007). Foram encontradas em seu computador mais de 200 imagens, contendo comandos de voz e texto ocultos, direcionadas ao controle do narcotráfico. Tendo as imagens temas infantis, sua presença no computador de um traficante levantou suspeitas. Foi necessária a ajuda do departamento antidrogas dos Estados Unidos (DEA - *Drug Enforcement Administration*) para extrair os dados, já que aliado à

esteganografia foi utilizada a criptografia (TERRA, 2008 e ZMOGINSKI, 2008). Essas notícias demonstram como a esteganografia pode ser perigosa quando utilizada para o crime.

1.4 Histórico

Um dos primeiros registros sobre a esteganografia é feito em “História”, de Heródoto (ALIGHERI et al, 1957), onde é descrito o uso de tábuas de escrita de madeira cobertas de cera (sobre as quais mensagens podiam ser escritas) e da cabeça de um escravo como meios de comunicação secreta. No primeiro método a cera era retirada e o texto a ser transmitido era escrito sobre a madeira. A cera retirada era então derretida novamente sobre a tábua com a mensagem, ocultando-a. Heródoto conta que Demerato, filho de Ariston, estava em Susa quando Xerxes decidiu liderar seu exército até a Grécia. Desejando secretamente avisar Esparta sobre o ataque, Demerato tomou duas destas tábuas, retirou a cera e escreveu sobre a madeira os planos de Xerxes. Feito isso derreteu a cera e cobriu novamente a tábua, ocultando a mensagem. Guardas que controlavam o tráfego e as cargas dos viajantes nas estradas nada viam, ao fiscalizar as encomendas, além de tábuas de escrita sem uso. Quando as tábuas chegaram à Grécia não havia ninguém que soubesse da existência e muito menos do método de revelação do segredo. Até o dia em que Gorgo, filha de Cleomenes e esposa de Leônidas, suspeitando das tábuas sem texto, retirou a cera e revelou as mensagens, possibilitando a Esparta se preparar para o ataque.

O segundo método consistia em raspar a cabeça de um escravo e tatuar a mensagem em sua cabeça. Assim que seu cabelo crescesse novamente o escravo era enviado sabendo somente como revelar a mensagem. Tal método era prático na época pois as notícias demoravam meses para serem transmitidas entre locais distantes.

Durante a história foram desenvolvidos vários outros métodos de ocultação de mensagens, a maioria destinada à comunicação durante guerras. Existem registros da Segunda Guerra Mundial sobre o uso de sucos de frutas cítricas ou de cebola, leite e vinagre como “tinta” para se escrever mensagens, sendo esta revelada ao se oxidar a solução utilizando-se uma fonte de calor (KAHN, 1996 apud BENTO; COELHO, 2004). Durante as duas Guerras Mundiais, os alemães utilizaram a tecnologia dos micropontos, que permitia que folhas inteiras de texto pudessem ser comprimidas ao tamanho de um ponto final comum ou o ponto de um “i” ou um “j”.

1.5 Princípios

O objetivo básico da esteganografia é a ocultação de uma mensagem em outra mensagem de cobertura, de modo que sua existência seja ignorada. Cole (2003) diz que três princípios centrais podem ser usados para medir a efetividade de um sistema de esteganografia:

- **Quantidade de dados:** a quantidade de dados que os sistemas de esteganografia permitem ocultar em um arquivo de cobertura pode ser utilizada para comparar suas eficiências.

- **Dificuldade de detecção:** define a dificuldade de se detectar a existência de dados ocultos após a utilização da técnica. Há uma relação direta, na maioria dos sistemas, entre quantidade de dados que podem ser ocultos e sua dificuldade de detecção: quanto maior a quantidade de dados inserida, maior é a distorção do arquivo de cobertura e conseqüentemente, a facilidade de se suspeitar da utilização da esteganografia.

- **Dificuldade de remoção:** característica que define que um possível interceptador do arquivo de cobertura não deve conseguir remover a mensagem (por extração ou destruição) facilmente.

1.6 Segurança

Nesta seção é apresentada a definição, baseada nos itens descritos por Cole (2003), das características de segurança de um sistema ideal de esteganografia. Segundo o autor, a segurança de computadores e de redes devem ter certos padrões que qualquer comunicação secreta deveria alcançar. Apesar de nenhum método alcançar completamente todos os requisitos, a esteganografia satisfaz muitos deles, algumas vezes em conjunto com outras técnicas como a criptografia. Na próxima sub-seção é feita a definição da criptografia e sua diferenciação da esteganografia e em seguida, as características definidas por Cole.

1.6.1 Criptografia

De acordo com Simon (1999, apud CHIARAMONTE et al):

> "A criptografia pode ser entendida como um conjunto de métodos e técnicas para cifrar ou codificar informações legíveis por meio de um algoritmo, convertendo um texto original em um texto ilegível, sendo possível mediante o processo inverso, recuperar as informações originais."

Existem dois métodos de criptografia: por código e por cifra. Na criptografia por código, as letras da mensagem são substituídas por códigos ou símbolos predefinidos, o que obriga o emissor e o receptor a manterem uma tabela de correspondência entre as letras e os códigos. Na cifragem, a transposição e a substituição eliminam a necessidade de manter essa correspondência, sendo necessário somente o conhecimento do processo de cifragem/decifragem. Neste método é utilizado o conceito de chaves.

Em transposição a mensagem é cifrada aplicando-se um desvio da posição de suas letras. Aplicando um desvio de quatro posições na palavra "CIFRA" sua cifragem resultaria na cifra "GMJVE".

Encriptação é a definição dada ao processo de transformar um texto legível em um texto ilegível, sendo a decriptação a transformação inversa. Para encriptar um texto utilizando a cifragem é necessário fornecer ao algoritmo um domínio de números ou símbolos que servirão para proteger a informação cifrada (CHIARAMONTE et al 2005). Esse domínio é chamado de chave. Somente com a utilização da chave é possível fazer a decriptação da mensagem.

O funcionamento de um algoritmo por cifragem é análogo ao de uma porta com fechadura. O funcionamento do algoritmo (que seria a fechadura) é público sendo a segurança do sistema baseado em sua chave. Foi provado historicamente que não existe um algoritmo que não possa ser quebrado (CHIARAMONTE et al, 2005), portanto seria imprudente confiar a segurança dos dados à restrição do conhecimento de seu funcionamento. A confiança do método é baseada no uso das chaves corretas.

As chaves criptográficas podem ser classificadas em simétricas ou assimétricas. Na criptografia de chave simétrica, a cifragem e a decifragem são feitos por meio de uma única chave, e na criptografia assimétrica, a cifragem é feita utilizando-se uma chave pública, conhecida por todos, e a decifragem por uma chave privada (gerada a partir da chave pública)

de conhecimento exclusivo do receptor da mensagem. Um modelo de sistema de criptografia de simétrica feito por Brunori Junior (1999) é apresentado na Figura 2.

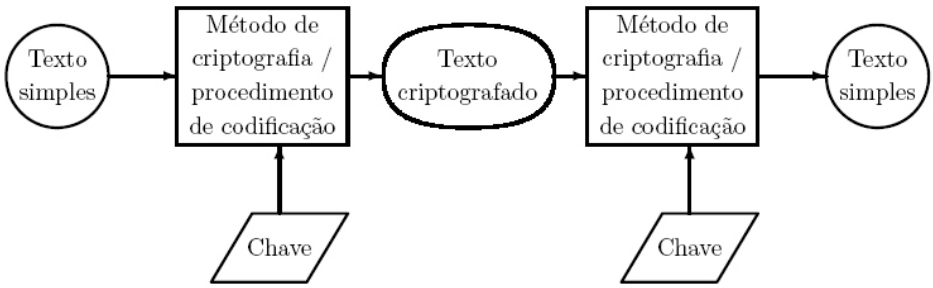


Figura 2 - Criptografia simétrica (Fonte: BRUNORI JUNIOR, 1999).

Embora façam parte de um mesmo domínio de estudo, a criptologia, os propósitos da esteganografia e da criptografia são distintos. Na criptografia, a mensagem é alterada de modo que seu significado seja ilegível e decifrável somente por meio de utilização de chaves e a segurança reside no sigilo das chaves utilizadas. Na esteganografia, a mensagem é incorporada ao objeto que a receberá, sendo a segurança confiada à manutenção das características deste objeto.

As duas técnicas podem funcionar em conjunto, uma como complemento da outra. Ao ocultar uma mensagem em uma imagem, pode-se aplicar a criptografia à ela e, posteriormente, embutir a mensagem cifrada. Se a imagem for interceptada, será necessário que o atacante descubra que existe uma mensagem oculta e, em seguida, como extrair essa mensagem. Após a extração será ainda preciso descobrir a chave utilizada para criptografá-la, podendo este processo, dependendo do tamanho da chave, demorar mais que o tempo de duração do universo (SCHNEIER, 1996 apud CHIARAMONTE et al, 2005).

1.6.2 Confidencialidade

Característica principal de um sistema de esteganografia, a confidencialidade da existência de informações ocultas em um objeto é essencial para que o método seja aplicável. Na maior parte das vezes que a esteganografia é aplicada, o objeto-recipiente trafega por meios de comunicação hostis onde qualquer suspeita levaria ao seu exame detalhado, facilitando a detecção, extração ou destruição da mensagem oculta. Portanto é crucial que o

objeto-recipiente seja de ocorrência freqüente no meio de comunicação utilizado. Nesta característica a esteganografia é mais eficiente que a criptografia.

1.6.3 Sobrevivência

A sobrevivência é a capacidade de um sistema de esteganografia de manter as características originais da mensagem oculta, após a transmissão e extração. Dados transmitidos por redes de comunicação podem sofrer alterações causadas por ruídos da linha e campos eletromagnéticos, entre outros motivos. Uma pequena mudança em um byte dos dados pode arruinar todo o processo de extração, dependendo da técnica utilizada na ocultação. A escolha do sistema de estenganografia a ser usado deve ser feita levando-se em consideração a qualidade do canal por onde o arquivo de cobertura trafegará.

1.6.4 Não-Detecção

É de pouca valia a utilização de um objeto que não seja suspeito se um interceptador puder facilmente detectar o uso da técnica e extrair a mensagem existente. Há softwares que vasculham todo o tráfego de uma rede buscando por arquivos que possam conter dados esteganografados, sendo irrelevante suas características visuais. Portanto além de um objeto bem escolhido, é necessário que o método de esteganografia empregado seja robusto.

Um método robusto é aquele em que o método de inserção é difícil de detectar e de destruir. Se um atacante descobre que há uma mensagem em um arquivo ele pode tanto tentar extraí-la quanto tentar destruí-la. Mesmo que o método utilizado seja conhecido, é necessário que seja forte o bastante para que a mensagem continue recuperável após uma tentativa de alteração do conteúdo oculto.

1.6.5 Visibilidade

Outro princípio básico da esteganografia, a visibilidade da mensagem oculta deve ser sempre nula. Um arquivo de áudio com muitos ruídos ou uma imagem de alta resolução que

tenha cores ou formas distorcidas evidencia que o arquivo sofreu algum tipo de alteração, tornando inútil qualquer algoritmo empregado caso a imagem sofra um ataque. Há uma troca entre visibilidade e capacidade de armazenamento de dados ocultos em um arquivo. Quanto maior a quantidade de dados ocultos mais perceptível é a existência de alterações do objeto-recipiente.

1.7 Considerações Finais

A esteganografia é resultado da necessidade de extremo sigilo que alguns tipos de arquivos devem ter ao serem armazenados ou transmitidos pela Internet. É uma técnica muito eficiente e de difícil detecção por software e mais difícil ainda visualmente. As possibilidades de uso são inúmeras, servindo para proteção de arquivos pessoais, marcação de direitos autorais, detecção de alteração de arquivos, crimes virtuais e reais, entre outros. Como levantado neste capítulo observa-se que, apesar de ainda não ter padrões bem definidos, como a criptografia, é muito promissora em diversas áreas da ciência de segurança de informações.

CAPÍTULO 2 - MÉTODOS DE ESTEGANOGRAFIA E SUAS CLASSIFICAÇÕES

Por não existir uma padronização da esteganografia como acontece com a criptografia, vários métodos foram criados, assim como definições para classificá-los. Alguns desses métodos e as classificações criadas para diferenciá-los são aqui estudados.

2.1 Métodos de Esteganografia

Os métodos de esteganografia definem como e onde os dados devem ser inseridos no arquivo de cobertura. Com o crescente estudo da esteganografia várias técnicas foram desenvolvidas e aplicadas, cada uma com sua abordagem particular. Em seguida são apresentados alguns dos tipos de esteganografia existentes, segundo Katzenbeisser e Petitcolas (2000).

2.1.1 Sistemas de Substituição

Sistemas deste tipo ocultam informações no arquivo de cobertura substituindo suas informações pelos bits dos dados sendo ocultados. São também chamados de ferramentas a nível de bit ou de inserção de ruído. São elas:

2.1.1.1 Substituição de Bit Menos Significativo

Na técnica LSB (*Least Significant Bit*, bit menos significativo) os bits menos significativos dos bytes do arquivo de cobertura são alterados para os bits da mensagem que se deseja ocultar. Utilizando uma substituição de 1 LSB são necessários 8 bytes da imagem de cobertura para cada byte a ocultar. A Tabela 1 demonstra o resultado final da inserção do caractere “A” em 8 bytes lidos de um arquivo de imagem. Os bits destacados são os que foram inseridos.

Tabela 1 - Processo de ocultação de um caractere usando a técnica LSB.

ASCII	Decimal	Hexadecimal	Binário
A	65	41	01000001

Decimal	Hexadecimal	Binário		Binário	Decimal
132	84	10000100	0	10000100	132
32	20	00100000	1	00100001	**33**
106	6A	01101010	0	01101010	106
50	32	00110010	0	00110010	50
135	87	10000111	0	10000110	**134**
68	44	01000100	0	01000100	68
161	A1	10100001	0	10100000	**160**
199	C7	11000111	1	11000111	199

A técnica permite que grande quantidade de dados sejam ocultados sem praticamente alterar perceptivelmente as características do arquivo original (KATZENBEISSER; PETITCOLAS, 2000).

2.1.1.2 Permutações Pseudo-Aleatórias

Técnica é similar à LSB, mas oculta os bits de forma diferente. Enquanto na técnica LSB os bits são embutidos nos bytes da cobertura sequêncialmente, nesta eles são espalhados em bytes escolhidos pseudo-aleatoriamente. Todos os bits da imagem de cobertura devem ser acessíveis dessa maneira. Esta técnica dificulta um ataque aos dados ocultos, pois não se sabe em qual byte da cobertura estão os bits a recuperar.

2.1.1.3 Degradação de Imagens e Canais de Cobertura

Duas imagens são utilizadas na aplicação desta técnica: uma de cobertura e a que será ocultada, ambas com as mesmas dimensões. Na inserção, para cada componente de cor da imagem de cobertura, seus 4 bits menos significativos são substituídos pelos 4 bits mais significativos da imagem sendo ocultada. Para extrair a imagem oculta, deve-se extrair os 4

bits menos significativos da estego-imagem e transformá-los em 4 bits mais significativos, que irão compor a imagem extraída.

2.1.2 Técnicas de Transformação de Domínio

Sistemas baseados em alteração de bits menos significativos podem sofrer alterações com ruídos do canal por onde trafegam ou por uma compactação com perdas da imagem de cobertura. Essas interferências podem destruir a integridade da mensagem oculta, tornando impossível sua recuperação. Com o desenvolvimento e evolução da esteganografia foi notado que é mais seguro armazenar os dados no domínio de frequência, sendo utilizado frequentemente a DCT (*Discrete Cosine Transform*, transformada discreta de coseno) na esteganografia em imagens.

Técnicas que utilizam a DCT operam em áreas significantes da imagem e mesmo assim as alterações ainda ficam imperceptíveis. Estego-imagens geradas com esta técnica são mais robustas em ataques de recorte e transformações dimensionais, podendo até serem compactadas com algoritmos com perdas e serem completamente recuperadas posteriormente. Apesar da maior segurança, há diminuição na quantidade de dados que podem ser inseridos na imagem pois as alterações são feitas em áreas, não em bytes individuais.

Além da DCT existem também a DFT (*Discrete Fourier Transform*, transformada discreta de Fourier), transformação *wavelet* e *Echo hiding* (para arquivos de áudio).

2.1.3 Espalhamento Espectral

Desenvolvida durante os anos 50 para aplicações militares, a técnica é utilizada contra interferências intencionais em radares (*anti-jamming*) e pela sua pouca probabilidade de interceptação. O espalhamento espectral é obtido quando se envia um sinal que ocupa uma largura de banda maior do que a necessária para a transmissão das informações (PICKHOLTZ et al, 1982 apud KATZENBEISSER; PETITCOLAS, 2000), que transforma o sinal em ruído. Esse espalhamento faz com que a energia do sinal em qualquer frequência seja baixa, dificultando sua detecção (OREBAUGH, 2004).

O uso do espalhamento espectral em esteganografia é feito ocultando em uma imagem de cobertura uma mensagem com ruído. A mensagem secreta é modulada de modo que se pareça com ruído Gaussiano. O sinal resultante, percebido como ruído, é embutido na imagem de cobertura, resultando na estego-imagem. Este método é robusto contra ataques de extração, mas ainda é vulnerável à perda dos dados por compressão ou processamento de imagem (OREBAUGH, 2004).

2.1.4 Observações Sobre os Métodos

Nota-se que ainda não existe um método completamente eficaz de esteganografia, sendo necessário escolher qual característica se deseja manter: confidencilidade, sobrevivência, não-detecção ou visibilidade. Quanto maior a satisfação de uma das características, maior a diminuição da eficiência quanto às demais.

A complexidade dos algoritmos aumenta cada vez mais, o que também aumenta a dificuldade de detecção, extração e destruição do conteúdo embutido.

2.2 Classificações da Esteganografia

Cole (2003) divide os métodos em duas classificações: original e nova. Na classificação original os métodos são classificados com base na maneira como os dados são ocultos e na nova é levado em consideração também onde os dados são ocultos. Diante do aparecimento de novas técnicas, a classificação original tornou-se insuficiente para abrangê-las, sendo criada a nova classificação.

2.2.1 Classificação Original

a) Baseado em Inserção

Algoritmos de esteganografia baseados em inserção procuram no arquivo de cobertura dados que podem ser alterados sem impacto significativo em suas características. Em todos os arquivos em que a técnica é aplicada os dados são inseridos sempre nos mesmos

locais. Dependendo do tipo de arquivo, os dados podem ser inseridos, por exemplo, entre cabeçalhos, tabelas de cores e bits menos significativos.

b) Baseado em Algoritmo

Métodos baseados em algoritmo elegem locais apropriados no arquivo de cobertura para ocultar os dados. Diferente dos algoritmos baseados em inserção, os dados não são sempre inseridos nos mesmos pontos, podendo causar degradação da qualidade do arquivo e tornar perceptível sua alteração.

c) Baseado em Gramática

Nesta classificação entram os métodos que geram seu próprio arquivo de cobertura. A partir de uma gramática pré-definida um arquivo de cobertura é criado, de modo que a saída se pareça o máximo possível com a gramática. Um exemplo seria a utilização de uma gramática que contenha palavras da seção financeiro de um jornal. Um algoritmo baseado em gramática geraria um texto de saída parecido com uma reportagem financeira. Quanto maior o nível (ou complexidade) da gramática, mais parecida com a linguagem especificada será a saída.

2.2.2 Nova Classificação

a) Baseado em Inserção

Os algoritmos de inserção da nova classificação procuram no arquivo de cobertura locais que são ignorados, pelo programa que o representa, quando é lido. Pode-se, por exemplo, inserir dados após a marcação de EOF (*End of file* ou fim de arquivo). Isso provoca aumento do tamanho do arquivo, que pode ser notado caso seja comparado ao arquivo de cobertura original. Nota-se que, embora estejam presentes nas duas classificações e tenham o mesmo nome, algoritmos baseados em inserção funcionam de forma distinta nas duas classificações.

b) Baseado em Substituição

Nesta abordagem se enquadram os algoritmos que substituem dados do aquivo de cobertura para ocultar dados, de forma que as alterações sejam imperceptíveis aos sentidos. Dessa maneira nem o tamanho nem as propriedades (visuais ou auditivas) do arquivo de cobertura são alterados. A quantidade de dados que pode ser oculta se limita à quantidade de dados existentes no arquivo de cobertura.

c) Baseado em Geração

Como nos métodos baseados em gramática, citados na classificação original, os métodos baseados em geração não necessitam de um arquivo de cobertura. Dessa maneira evita-se que o arquivo de cobertura original seja comparado a um que tenha os dados ocultos, podendo levar à detecção da utilização da esteganografia. A técnica mais utilizada em algoritmos baseados em geração é a criação de imagens fractais. As imagens são geradas fornecendo os dados a serem ocultos como semente para a fórmula matemática utilizada.

2.3 Considerações Finais

Ao final deste Capítulo percebe-se o quanto a esteganografia é estudada atualmente e a evolução dos métodos, de simples substituições de bits a complexas fórmulas de transformação de domínio. As técnicas LSB são vantajosas em quantidade de dados que se pode ocultar em um arquivo de cobertura e a facilidade de implementação mas têm a desvantagem de serem sucetíveis a ruídos, compactação e processamento do arquivo de cobertura. As técnicas de transformação discreta e espalhamento de espectro mantêm a informação mais protegida, porém suas eficiências de armazenamento são reduzidas.

A criação de tanto material novo levou à criação de parâmetros de classificação que, embora informais, servem como referência enquanto não é feita uma padronização oficial.

CAPÍTULO 3 - COMPACTAÇÃO DE DADOS

3.1 Introdução

A compactação de dados é uma técnica criada para reduzir a quantidade de caracteres necessários para se representar uma mensagem, eliminando redundâncias, e permitir sua recuperação posterior. Entretanto não se deve confundir compactação com compressão. Na compactação, as redundâncias são reduzidas sem perdas de dados, enquanto que na compressão, dados são perdidos e não podem ser mais recuperados (SAADE, 2008). É usada, por exemplo, em arquivos de áudio e vídeo que não precisam reproduzir detalhes que os sentidos humanos não percebem.

Entretanto a compactação e a compressão de dados não são usadas somente em arquivos. Técnicas foram desenvolvidas para transmissão de sinais de televisão, tráfego de dados em uma rede (ARRUDA; GOES, 2003), telefonia, bancos de dados, entre outros. Dentre os algoritmos existentes, o algoritmo de Huffman foi escolhido para ser abordado neste trabalho, por sua eficiência e por servir de base para importantes algoritmos de compressão como o JPEG. Coello (1994) faz uma comparação entre vários algoritmos de compactação e apresenta as Tabelas 2 e 3. Percebe-se que os algoritmos de Huffman têm a melhor relação entre razão de compressão (coluna T/L[1]) e tempo para arquivos grandes, como mostra a tabela superior, e que para arquivos pequenos perdem no mesmo quesito somente para o algoritmo LZW 12 bits. Com isso, foi escolhido o algoritmo de Huffman como objeto de estudo.

[1] *T* é o tamanho do arquivo original e *L* o tamanho do arquivo compactado, ambos em bytes.

Tabela 2 - Comparativo entre vários algoritmos de compactação de dados aplicados a um arquivo .doc (Fonte: COELLO, 1994).

Método	Tam. Original[2]	Tam. Comp.	T/L	Razón de Comp.	100 Log_e(T/L)	Tiempo (seg)
Huffman	13224063	4725562	2.80	64.27%	102.91	118.96
Huffman-A[3]	13224063	2691246	4.91	79.65%	159.20	174.48
Cod. Aritm 0[4]	13224063	4651746	2.84	64.82%	104.48	552.99
LZSS	13224063	2364813	5.59	82.12%	172.13	460.05
LZW 12 bits	13224063	18141837	0.73	-37.19%	-31.62	314.45
Diferencial	13224063	4595543	2.88	65.25%	105.70	816.24
Sixpack	13224063	N.D.	N.D.	N.D.	N.D.	N.D.
Cod. Aritm 1[5]	13224063	876789	15.08	93.37%	271.35	4781.54
Mix-1 0	13224063	2118179	6.24	83.98%	183.15	25212.23

Resultados produzidos ao aplicar os métodos em estudo a um arquivo em formato Microsoft Word para Windows 2.0. Os testes ocorreram em um computador 486 DX/2 de 66MHz com coprocessador matemático compatível com IBM. O método **Sixpack**, que consiste em uma mistura de 6 técnicas diferentes, produziu um erro de transbordamento. O método **Mix-1** também é um híbrido que se provou unicamente ao nível zero por sua grande ineficiência.

Tabela 3 - Comparativo entre vários algoritmos de compactação de dados aplicados a um arquivo .dbf (Fonte: COELLO, 1994).

Método	Tam. Original[6]	Tam. Comp.	T/L	Razón de Comp.	100 Log_e(T/L)	Tiempo (seg)
Huffman	504802	180317	2.80	64.28%	102.95	4.01
Huffman-A[7]	504802	169595	2.98	66.40%	109.08	7.69
Cod. Aritm 0[8]	504802	176729	2.86	64.99%	104.95	20.76
LZSS	504802	141623	3.56	71.94%	127.10	26.80
LZW 12 bits	504802	133938	3.77	73.47%	132.68	5.16
Diferencial	504802	312909	1.61	38.01%	47.83	53.28
Sixpack	504802	85210	5.92	83.12%	177.90	16.92
Cod. Aritm 1[9]	504802	122461	4.12	75.74%	141.64	107.71
Mix-1 0	504802	164523	3.07	67.41%	112.11	12950.99

Resultados produzidos ao aplicar os métodos em estudo a um arquivo em formato .dbf. Os testes ocorreram em um computador 486 DX/2 de 66MHz com coprocessador matemático compatível com IBM.

[2] Em bytes
[3] Huffman Adaptativo
[4] Modelo de ordem zero feito usando codificação aritmética
[5] Modelo adaptativo de ordem n com codificação aritmética de ordem 1
[6] Em bytes
[7] Huffman Adaptativo
[8] Modelo de ordem zero feito usando codificação aritmética
[9] Modelo adaptativo de ordem n com codificação aritmética de ordem 1

3.2 Algoritmo de Huffman

O algoritmo de Huffman foi desenvolvido por David Huffman em 1950 (BLELLOCH, 2001) e consiste na utilização de uma árvore binária para geração dos

chamados códigos de Huffman para compactação de dados. Por ser simples e eficiente serve como base para softwares compactadores de arquivos e para algoritmos de compactação de imagens (GZIP e JPEG, respectivamente). Os código de Huffman são códigos binários criados a partir da frequência que um símbolo aparece na origem dos dados. Quanto maior a frequência de um símbolo menor deve ser seu código, sendo que um código nunca é prefixo de outro (CAMPOS, 1999). As duas formas de implementação do algoritmo são detalhadas a seguir.

3.2.1 Huffman Estático

Para compactar um arquivo utilizando Huffman estático são necessárias duas leituras do arquivo: uma para calcular as frequências dos símbolos e outra para fazer a compactação. Durante a primeira leitura é criada uma tabela com os símbolos e suas frequências. Com estes dados uma árvore binária de busca é criada, em que cada nó folha representa um símbolo. Quanto mais próximo da raiz está o símbolo, menor é o código atribuído a ele. Nesta árvore somente os nós folha armazenam os símbolos enquanto os nós internos recebem a soma das frequêncis de seus nós filhos.

Na segunda leitura, para cada símbolo lido é feita sua busca na árvore e o caminho da raiz até ele é escrito na saída dos dados. Supondo que se deseja compactar a sequência de caracteres “abaacba”, o processo seria:

a) Ler os dados a primeira vez e calcular as frequências (Tabela 4).

Tabela 4 - Frequências dos símbolos no algoritmo de Huffman estático.

Símbolo	Quantidade	Frequência
a	4	4/7 = 0,5815
b	2	2/7 = 0,2856
c	1	1/7 = 0,1429

a) Criar a árvore binária de Huffman. Com a lista de frequências *Lf*, o algoritmo para criá-la pode ser escrito como na Figura 3:

```
Enquanto houver mais de um símbolo na lista faça
        Encontrar os dois símbolos de menor frequência (s1 e s2) de Lf;
        Criar dois nós (n1 e n2), um para cada símbolo encontrado;
        Retirar os símbolos s1 e s2 da lista;

        Criar um nó pai np para os nós n1 e n2;
        Se (frequência n1 >= frequência n2)
                Filho direito de np= n1;
                Filho esquerdo de np = n2;
        Senão
                Filho direito de np= n2;
                Filho esquerdo de np = n1;
        Fim senão;

          Frequência de np = frequência n1 + frequência n2;
          Inserir np na lista;
Fim enquanto;
```

Figura 3 - Algoritmo de criação da árvore de Huffman estático.

A Figura 4 ilustra a árvore final para a tabela do exemplo.

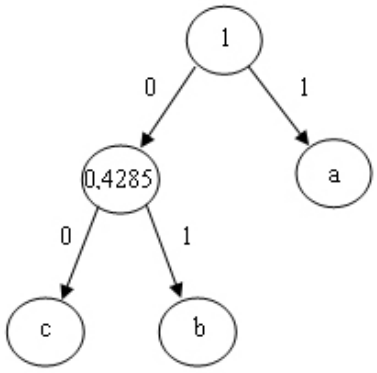


Figura 4 - Exemplo de árvore de Huffman estático.

b) Após a criação da árvore, volta-se ao início dos dados. Todos os símbolos são lidos novamente, agora escrevendo o caminho na árvore da raiz até ele, sendo “1” para percurso à direita e “0” para percurso à esquerda. O resultado desta etapa são os códigos da Tabela 5.

Tabela 5 - Códigos gerados pelo algoritmo de Huffman estático.

Símbolo	Código
a	1
b	01
c	00

Sequência de caracteres “abaacba” compactada: 1011100011

Para representar a sequência de caracteres original seriam necessários 56 bits sendo 8 bits para cada caractere. Após a compactação a quantidade foi reduzida para 10 bits, ou seja, houve uma diminuição de 82,15% na quantidade de bits necessários.

Para a descompactação deve ser fornecido ao algoritmo a tabela de símbolos e frequências que gerou a árvore de Huffman na compactação. A árvore é então recriada e a partir da sequência de bits dos dados compactados, os caracteres são extraídos.

O algoritmo de descompactação é mostrado na Figura 5, Sendo *F* a fonte de códigos de Huffman (lida da esquerda para a direita), *Ah* a árvore recriada, *n* um ponteiro para os nós da árvore e *Sd* a saída dos dados descompactados.

```
n aponta para o nó raiz de Ah;

Enquanto houver bits em F
        Lê um bit de F;
        Se bit é 0 n aponta para seu filho esquerdo;
        Senão n aponta para seu filho direito;
        Fim senão;

        Se n é um nó folha
                Escreve o símbolo em n em Sd;
                n aponta para o nó raiz de Ah;
        Fim se;
Fim enquanto;
```

Figura 5 - Algoritmo de descompactação de Huffman estático.

Apesar da grande redução dos dados, o algoritmo de Huffman estático tem duas grandes desvantagens: ter de ler o fonte de dados duas vezes, uma para calcular as frequências e outra para comprimir o arquivo, e ter que enviar a tabela de símbolos e suas respectivas frequências junto com os dados compactados, reduzindo muito a eficiência final do algoritmo (LOW, 2000). Enviando os símbolos, a quantidade de vezes que cada um aparece e o resultado final da compactação tem-se a Equação 1.

Equação 1 - Total de bits necessários para enviar dados compactados por Huffman estático.

$$Tb = (Qs \times 40 \text{ bits}) + Bc;$$

Sendo:

Tb: Total de bits;

Qs: quantidade de símbolos diferentes lidos;

40 bits: soma de 1 byte (1 char) + 4 bytes (1 int) para a quantidade de vezes que aparece;

Bc: Quantidade de bits compactados.

Para o exemplo dado seriam necessários então *Tb* = (3x40) + 10, resultando em um total de 130 bits, excedendo em 132,15% a quantidade de bits originalmente necessária.

3.2.2 Huffman Adaptativo

Dadas as desvantagens do Huffman estático, Faller e Gallagher, e mais tarde Knuth e Vitter, desenvolveram um modo de executar o algoritmo de Huffman como um procedimento de uma passagem pelos dados de origem (SAYOOD, 2000 apud LOW, 2000). Neste novo algoritmo a criação da árvore e a geração dos códigos são feitos durante a leitura da fonte dos dados. À medida que novos símbolos são lidos, a árvore é remodelada, aumentando a efetividade da compactação. Aqui é explicado o método de Vitter.

Em Huffman estático se um símbolo aparece somente uma vez, no início de um arquivo, ele será colocado em níveis mais baixos da árvore, recebendo um código com vários bits. Em Huffman adaptativo esse símbolo é colocado primeiro em níveis mais altos e recebe um código curto, e com a leitura de mais símbolos é deslocado para níveis mais baixos.

3.2.2.1 Propriedades da Árvore de Huffman Adaptativo

A explicação feita a seguir é baseada em Low (2000). Cada nó de uma árvore de Huffman adaptativo deve conter o símbolo que armazena, seu peso (quantidade de vezes que foi lido), e a ordem do nó na árvore. Deve ainda ter três ponteiros: para o nó pai, para o nó filho esquerdo e para o nó filho direito. Há um nó especial chamado NYT (*Not Yet Transmitted*, ainda não transmitido) que deve estar presente na árvore desde sua criação, pois é onde novos símbolos são inseridos. Observando a Figura 6 nota-se as propriedades da árvore que devem ser mantidas enquanto ela é atualizada:

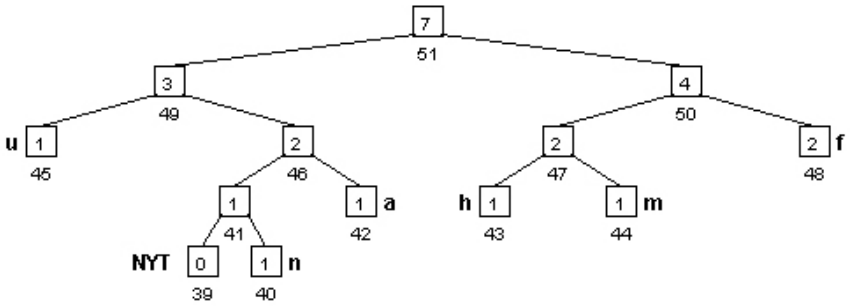


Figura 6 - Árvore[2] de Huffman adaptativo resultante da compactação da palavra "huffman".

1) Todo nó tem um nó irmão;
2) Nós com maiores pesos têm maiores ordens;
3) Em cada nível da árvore o nó mais à direita tem a maior ordem, embora possam haver outros nós com mesmo peso em um mesmo nível;
4) Nós folha sempre armazenarão símbolos, exceto o nó NYT;
5) Nós internos não armazenam símbolos e têm peso igual à soma dos pesos de seus dois filhos;
6) Nós que têm o mesmo peso devem ter ordens consecutivas.

A manipulação da árvore é mais complexa do que em Huffman estático, dadas as várias regras que devem ser satisfeitas. Quando criada, a árvore contém somente o nó NYT, que é também a raiz da árvore. Ao receber um novo símbolo é feita sua procura na árvore. Se não foi encontrado, o nó NYT é expandido em dois nós. O nó à direita contém o novo símbolo e o nó à esquerda se transforma no novo nó NYT. O antigo nó NYT é então transformado em um nó interno, recebendo peso 1 e mantendo sua ordem. A inserção do caractere "u" é ilustrado na Figura 7.

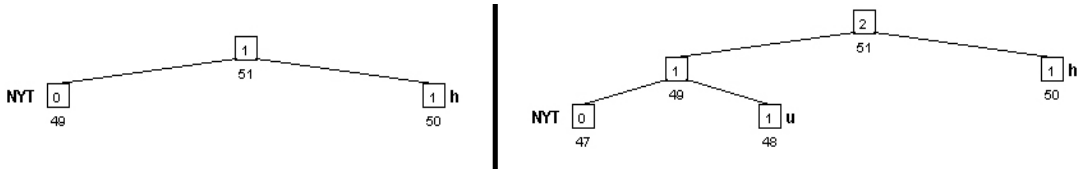


Figura 7 - Expansão do nó NYT.

[2] Árvore criada utilizando o *applet* Java de Dominik Szopa, disponível em http://www.cs.sfu.ca/cs/CC/365/li/squeeze/AdaptiveHuff.html.

Se o símbolo já existir, o peso de seu nó é incrementado. Antes porém, deve ser verificado se o nó sendo atualizado é o de maior ordem da classe de peso. Se não é, deve ser trocado com o nó de mesmo peso mas de maior ordem da classe de peso. A Figura 8 mostra uma árvore que contém os caracteres “huf” na qual é inserido outro caractere “f”.

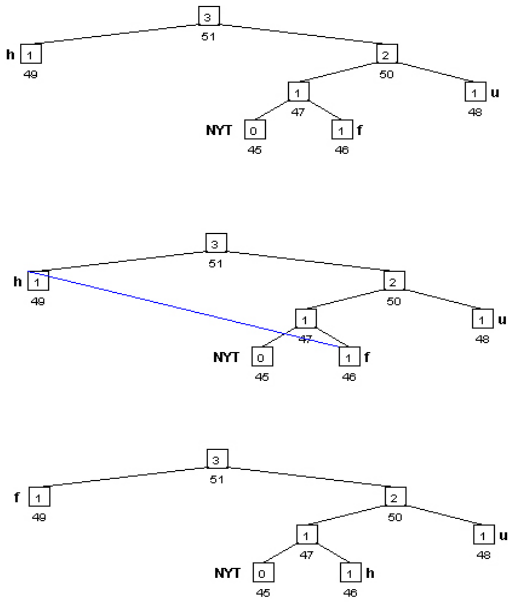


Figura 8 - Troca de nós durante atualização da árvore.

Em ambos os casos, de existência ou não do símbolo na árvore, a mudança do peso de um nó folha afeta os pesos de todos os nós acima dele. A partir do nó que foi alterado ou inserido, deve ser conferido se o nó acima dele (seu nó pai) é o de maior ordem em sua classe de peso. Se não for, deve-se seguir o procedimento de troca de nós descrito no parágrafo anteriormente. Somente após realizado este procedimento é feita a atualização do peso do nó. A conferência é sempre feita subindo árvore, até chegar ao nó raiz. Na troca de nós também há regras que devem ser seguidas:

1) A raiz nunca é trocada com nenhum outro nó;
2) Um nó nunca é trocado com seu nó pai;
3) Ao fazer a troca de nós, trocar também suas ordens. A ordem não tem relação com o peso ou o símbolo de um nó, mas sim com o lugar ocupado por ele na árvore.

A complexidade de manipulação da árvore é compensada pela facilidade de codificação e decodificação da origem dos dados.

3.2.2.2 Codificação

Por ser um algoritmo de uma passagem pelos dados, a manipulação da árvore e a codificação são feitas em conjunto. Ao ler um símbolo já existente o caminho da raiz até ele deve ser escrito na saída dos dados (um arquivo, por exemplo), sendo “1” para percurso à direita e “0” para a esquerda. Se é um novo, primeiro é escrito o caminho até o nó NYT, sinalizando uma inserção e em seguida, o novo símbolo é escrito na saída dos dados. Para ambos os casos o caminho deve ser escrito antes da atualização da árvore. O procedimento é repetido até o final da origem dos dados.

Utilizando a mesma sequência de caracteres de exemplo da forma estática, o resultado final de sua compactação pelo método adaptativo é a sequência “a0b1100c011”. Convertendo os caracteres em bits (sublinhados) temos “01100001001100010110001100011011” = 32 bits de dados compactados contra os 56 originais, resultando numa diminuição de 42,85%. A Figura 9 mostra a árvore ao final do processo.

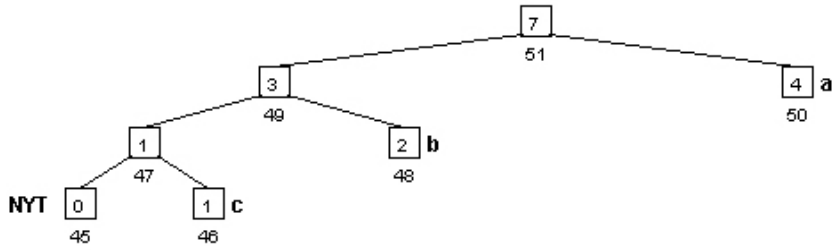


Figura 9 - Árvore de Huffman adaptativo resultante da compactação da sequência de caracteres “abaacba”.

3.2.2.3 Decodificação

Na decodificação não é necessário nenhum conhecimento prévio dos dados que serão lidos, pois a leitura dos dados compactados e a decodificação ocorrem simultaneamente. Também não é necessário fazer buscas de símbolos na árvore, pois a própria fonte de dados compactados “guia” o descompactador até eles. Temos na Figura 10 o algoritmo de

decodificação, sendo *F* a fonte de dados compactados, *Aha* a árvore recriada, *n* um ponteiro para os nós da árvore e *Sd* a saída dos dados descompactados.

```
n aponta para a raiz da árvore;
Enquanto houver dados em F
        Se n aponta para o nó raiz
                Lê 8 bits de F (um caractere);
                Escreve o caractere em Sd;
                Insere o novo caractere em Aha e a atualiza;
        n aponta para a raiz da árvore;
        Senão
                Enquanto n apontar para um nó interno
                        Lê 1 bit de F;
                        Se bit é 0 n aponta para seu filho esquerdo;
                        Senão n aponta para seu filho direito;
                Fim enquanto;

                Se o nó apontado por n não é o nó NYT
                        Escreve o caractere do nó apontado por n em Sd;
                        n aponta para a raiz da árvore;
                Fim se;
        Fim senão;
Fim enquanto;
```

Figura 10 - Algoritmo de decodificação de Huffman adaptativo.

Portanto, apesar da complexidade de manipulação da árvore, o resultado final é melhor que o do algoritmo de Huffman estático por não precisar manter tabelas de frequência ou armazenar qualquer dado junto com o códigos criados, além dos próprios caracteres.

3.3 Considerações Finais

Os algoritmos de Huffman são eficientes e simples de implementar, tendo várias aplicações práticas. Embora o algoritmo de Huffman estático seja menos eficiente no resultado final da compactação, sua velocidade é bem maior que os outros algoritmos testados por Coello (1994). Já o modelo adaptativo consegue melhores resultados por não precisar previamente dos símbolos que serão compactados, apesar de ser um pouco mais lento. Tem ainda as vantagens de adaptabilidade dos códigos gerados e de servir em aplicações onde o fluxo de entrada de dados é constante.

CAPÍTULO 4 - DESENVOLVIMENTO DE UM SISTEMA DE ESTEGANOGRAFIA

Com o conjunto de técnicas e algoritmos estudados nos Capítulos anteriores, é dado início ao desenvolvimento do sistema proposto.

4.1 Introdução

Neste trabalho é proposto o desenvolvimento de um sistema de esteganografia em imagens, aliada à compactação de dados. Essa abordagem difere da comum, que é o uso da criptografia e coincide com a proposta de Rocha (2003, p 79) do uso de um algoritmo de Huffman em conjunto com a esteganografia. A intenção com a compactação dos dados é aumentar a capacidade de ocultação do arquivo de cobertura e dificultar a detecção do uso da técnica.

O sistema foi desenvolvido em linguagem C, sendo composto de duas bibliotecas e um programa principal, que as implementa. É executado por linha de comando, fornecendo os parâmetros necessários para cada tipo de operação. A compactação de arquivos e a esteganografia foram implementadas como bibliotecas estáticas C, para permitir avaliação individual e reutilização do código. Todas as funções foram desenvolvidas neste projeto, exceto as já existentes das bibliotecas padrão. Para o desenvolvimento foi utilizado o Bloodshed Dev-C++ versão 4.9.9.2. As técnicas e formatos escolhidos para compor o sistema são:

- Arquivo de origem dos dados: O sistema aceita qualquer formato de arquivo a ser oculto, pois sua estrutura não é relevante no processo de ocultação.

- Algoritmo de compactação: O algoritmo de compactação de Huffman adaptativo foi escolhido para este trabalho por sua eficiência e adaptabilidade.

- Ocultação: A técnica de esteganografia LSB foi escolhida por seu funcionamento simples e eficiente e pela facilidade de visualizar seu funcionamento.

- Arquivo de cobertura: O formato de imagem bitmap foi escolhido por sua estrutura simples, permitindo melhor entendimento do processo de ocultação e pelo tamanho dos arquivos, podendo ocultar grande quantidade de dados.

O nome escolhido para o sistema foi *WaxTablet*, em referência ao uso de tábuas de escrita utilizadas na Grécia, como foi apresentado no histórico da esteganografia, no Capítulo 1.

4.2 Objetivos

O objetivo do sistema é demonstrar o funcionamento da esteganografia aliada à compactação de dados, mantendo o tamanho do arquivo de cobertura inalterado e permitindo a extração completa dos dados ocultos sem distorções ou falhas. Após a ocultação e extração são apresentadas informações de desempenho para avaliação dos algoritmos implementados.

As bibliotecas desenvolvidas também fazem parte do resultado final do trabalho e podem ser utilizadas em outros estudos ou por outros desenvolvedores.

4.3 Estruturação Geral

O sistema é composto de duas bibliotecas estáticas C e um programa principal, que as implementa. As bibliotecas desenvolvidas funcionam independentemente para que possam ter seus desempenhos medidos e serem reutilizadas. As partes que compõem o sistema são:

- Biblioteca adaphuff: Implementa a compactação de arquivos, gerando um arquivo “.ahf” com os dados compactados.

- Biblioteca stglsb: Implementa a inserção dos dados de um arquivo nos bits menos siginificativos das cores da imagem bitmap de cobertura.

- Programa principal: Implementação de um sistema de esteganografia usando as duas bibliotecas criadas.

Em seguida são apresentadas as estruturas de dados utilizadas e os principais problemas encontrados durante o desenvolvimento das bibliotecas e qual solução foi adotada para contorná-las

4.3.1 Biblioteca adaphuff

É a biblioteca responsável pela implementação da compactação de dados por Huffman adaptativo. A estrutura de dados, em C, que implementa sua árvore está na Figura 11.

```
//Estrutura dos nós da árvore.
typedef struct no
{
        unsigned short caractere;
        unsigned int peso;
        unsigned int ordem;

        struct no *esquerdo;
        struct no *direito;
        struct no *pai;

} no;

//Estrutura da árvore de Huffman, com o vetor auxiliar
//e a raiz da árvore.
typedef struct AHA
{
        no *raiz;
        no *vetor[ORDEM_MAXIMA];
} AHA;
```

Figura 11 - Estrutura de dados em C da árvore de Huffman adaptativo implementada.

Todas as variáveis que compõem a estrutura *no* não armazenam valores negativos. A variável *caractere* teve de ser alterada de *unsigned char* para *unsigned short*, para que o compactador funcionasse com arquivos executáveis. O problema apareceu quando caracteres não-imprimíveis eram inseridos na árvore. Como o nó NYT foi inicialmente definido tendo valor de caractere “0” decimal, era criado um conflito ao ler um caractere com esse valor de

um arquivo executável e fazer sua inserção. Além do nó NYT foi necessário encontrar uma representação para os nós internos e para o nó PSEUDO_EOF, que não consta no algoritmo original mas que foi criado neste projeto para sinalizar ao descompactador o final dos dados, pois ao extrair os dados compactados de uma imagem, bits a mais podem ser e influenciar na descompactação. A solução foi fazer, no cabeçalho da biblioteca, a definição de constantes que armazenariam os números reservados da estrutura da árvore (Figura 12).

```
#define NYT 0
#define NO_INTERNO 1
#define PSEUDO_EOF 2
#define DESLOCAMENTO 3
#define ORDEM_MAXIMA 512 + DESLOCAMENTO
```

Figura 12 - Declarações de constantes específicas da implementação de Huffman adaptativo feita.

Essa declaração reserva os decimais de 0 a 2 para os nós específicos da implementação da árvore. Dessa maneira são necessários pelo menos 256 (caracteres da tabela ASCII) + 3 = 259 valores para que a árvore possa armazenar todos os caracteres mais seus nós específicos. Como o valor 259 excede o máximo suportado pelo tipo *unsigned char* (1 byte), a solução foi declarar *caractere* como *unsigned short* (2 bytes), já que não há nenhum tipo de dado inteiro entre os dois tipos. Assim, o valor do identificador *DESLOCAMENTO* é aplicado ao caracetere lido do arquivo sendo compactado e então é inserido na árvore. Na descompactação o processo é inverso, sendo retirado o deslocamento para que o caractere seja corretamente recuperado. Foram feitos testes com arquivos executáveis no compactador e após a extração, continuavam funcionando corretamente.

As variáveis *peso* e *ordem* foram definidas como *unsigned int*, o que dá um alcance de valor decimal de 0 até 4.294.967.295, ou seja, 4 gigabytes da dados, mais do que suficiente para qualquer arquivo comum. São definidos também três ponteiros para estruturas do tipo *no*, que são usadas para manter as referências aos nós filhos e pai de cada nó.

Junto com a árvore foi implementado um vetor, seguindo a proposta de Low (2008); um vetor de ponteiros para os nós da árvore. Assim é feita uma busca linear no vetor (embora não seja a mais eficiente) ao invés de constantes buscas recursivas na árvore. Foi então criada a estrutura *AHA* (Árvore de Huffman adaptativo) que contém dois ponteiros: um para o vetor auxiliar e outro para a raiz da árvore.

Houve ainda um problema quanto à recuperação da extensão original do arquivo compactado. Para solucionar o problema, na compactação, antes de escrever os dados compactados, os caracteres da extensão são escritos no início do arquivo, mas com a ordem dos seus bits invertidos, alterando o caractere.

4.3.2 Biblioteca stglsb

Para o desenvolvimento desta biblioteca não foi necessária a criação de estruturas de dados específicas, mas sim o estudo da estrutura de arquivos bitmap. O formato bitmap foi desenvolvido para ser o formato nativo de imagens do sistema operacional Windows, a partir da versão 3.0. A estrutura do arquivo é bem simples para evitar erros de leitura ou inconsistências (OLIVEIRA, 2000), e consiste de 2 cabeçalhos, uma paleta de cores (opcional) e da área de imagem (Figura 13). Por causa de sua estrutura simples de cabeçalho e de área de dados de imagem, e pela grande quantidade de bytes, sem compactação, que a compõe, foi o formato escolhido para as imagens de cobertura. Os cabeçalhos armazenam informações sobre o arquivo como suas propriedades de tamanho e profundidade de cores. As definições de cada um são descritas em (OLIVEIRA, 2000).

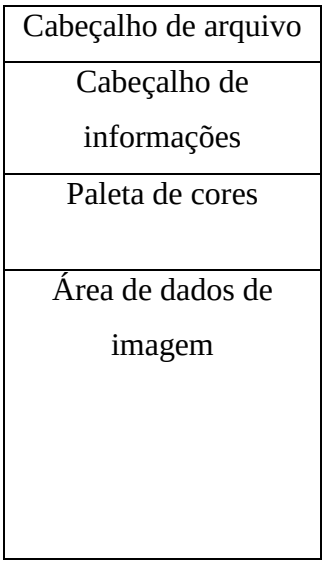


Figura 13 - Estruturas de um arquivo bitmap (Fonte: BOURKE, 1998)

A paleta de cores somente está presente em imagems com 256 cores ou menos. Como imagens bitmap com menos de 24 bits de profundidade são muito sensíveis a alterações

de suas cores, não são consideradas pelo sistema. O tamanho do cabeçalho de arquivo mais o cabeçalho de informações é de 54 bytes. Como essas estruturas não têm relevância durante a ocultação de dados, são simplesmente copiadas de um arquivo pra outro (na ocultação) ou ignoradas (na extração). A parte que interessa é área de dados imagens.

Em imagens bitmap de 24 bits cada pixel (*picture element*) é composto de 3 bytes, um para cada cor primária, isto é, vermelho, verde e azul. A combinação desses três valores determina a cor que o pixel exibirá. Deve-se notar que o primeiro pixel da área de dados corresponde ao canto inferior esquerdo da imagem, e o último ao canto superior direito.

É nos componentes de cor de cada pixel que os bits da mensagem são ocultos. Alterando o bit menos significativo não há qualquer mudança aparente na imagem. Dependendo da imagem as alterações só passam a ser visíveis a partir da substiuição de 3 ou 4 bits menos significativos.

Mesmo não necessitando de estruturas de dados específicas, foi necessária a criação um cabeçalho de dados referentes ao processo de ocultação. Neste cabeçalho, inserido antes dos dados em si, constam a quantidade de bits menos significativos que foram substituídos, a extensão original do arquivo oculto e a quantidade de bits a ler. Como não é possível prever a quantidade de LSBs usados, são necessários sempre 3 bytes para armazená-la, substituindo somente 1 LSB de cada byte. Os decimais de 1 a 8 são decompostos em bits, sendo utilizados somente os 3 bits menos significativos resultantes. Embora a decomposição do número decimal 8 resulte na sequência de bits “00001000”, o que necessitaria de 4 bits, seus últimos 3 bits resultariam no número 0. Como este número não está entre os possíveis, sempre que a quantidade de LSBs usada for 0, é convertida no decimal 8. A Tabela 6 ilustra essa conversão.

Tabela 6 - Conversão dos bits que representam a quantidade de LSBs substituídos em uma estego-imagem.

Quantidade de LSBs substituídos	Binário	Bits lidos pelo extrator	Quantidade convertida
1	**00000001**	**001**	**1**
2	**00000010**	**010**	**2**
3	**00000011**	**011**	**3**
4	**00000100**	**100**	**4**
5	**00000101**	**101**	**5**
6	**00000110**	**110**	**6**
7	**00000111**	**111**	**7**
8	**00001000**	**000**	**8**

Após a quantidade de LSBs usados é escrita a extensão do arquivo original, sem o ponto e a quantidade de bits a ler. As duas informações ocupam respectivamente 3 e 4 bytes (3 *chars* mais um *int*), sendo necessários 7 bytes dividido pela quantidade de LSBs usados. Caso sobrem bits a substituir no último byte (o que acontece para valores de LSB 3, 5 e 6) os restantes são preenchidos com zero. Isso evita que bits da mensagem e do cabeçalho se misturem, dificultando a recuperação. Após escrito o cabeçalho, o restante dos bits é escrito, respeitando a quantidade de LSBs a substituir. Na extração esse cabeçalho é lido e o arquivo é recuperado seguindo as instruções gravadas.

4.4 Serviços Implementados

Os serviços implementados neste projeto são a compactação de dados por Huffman adaptativo e a ocultação de arquivos em imagens bitmap por esteganografia utilizando a técnica LSB. Os protótipos das funções e o que fazem são:

- Biblioteca adaphuff
    - char* compactar(char *nomeArquivoOrigem)

O compactador recebe um nome de arquivo como parâmetro e gera como saída um arquivo compactado de nome igual ao original adicionado do sufixo “_compactado”, e com extensão “.ahf”. A função retorna um ponteiro para o nome do arquivo criado.

- void descompactar(char *nomeArquivoCompactado)

Da mesma maneira, para descompactar, deve-se fornecer o nome de um arquivo ".ahf" e após feita a descompactação o arquivo original é recriado, com a adição do sufixo "_descompactado" e com sua extensão original.

- Biblioteca stglsb
  - void ocultarLSB(char *nomeArquivoOrigem, char *nomeImagemOrigem)

Para ocultar um arquivo em uma imagem é necessário passar à ferramenta de esteganografia dois parâmetros: o arquivo que será oculto e a imagem que servirá de cobertura. O processo cria um arquivo de mesmo nome da imagem, adicionado do sufixo "_estego" para diferenciar da imagem original.

  - char* extrairLSB(char *nomeImagemOrigem)

Para extrair dados esteganografados, deve-se passar como parâmetro somente a estego-imagem da qual se deseja extrair os dados ocultos. Após a extração será recriado o arquivo que estava oculto, de mesmo nome da estego-imagem, adicionado do sufixo "_extraido" e com sua extensão original. A função retorna um ponteiro para o nome do arquivo criado.

O modo que o programa principal deve ser executado e seu funcionamento é:

- WaxTablet.exe –o arquivo1.xxx imagem.bmp

Para ocultar um arquivo em uma imagem, após o nome do executável o parâmetro "-o" deve ser fornecido, seguido do nome do arquivo que se deseja ocultar e da imagem que servirá de cobertura. O programa faz a compactção do arquivo e depois insere o arquivo criado pela compactação na imagem de cobertura.

- WaxTablet.exe –e imagem.bmp

Para fazer a extração de dados de uma imagem, deve ser fornecido o parâmetro "-e" seguido do nome da imagem que contém os dados. O programa faz a extração dos dados, seguido de sua descompactação, gerando o arquivo original.

4.5 Considerações Finais

Sendo terminado o estudo e a estruturação do sistema, foi feita sua implementação. Foram desenvolvidos protótipos de cada biblioteca que foram testados e corrigidos várias vezes até sua versão final. Após suas validações finais, o programa principal que as implementa foi escrito e testado, chegando ao WaxTablet 1.0. Os objetivos estabelecidos foram alcançados e testes de desempenho foram feitos, que são apresentados no próximo Capítulo.

CAPÍTULO 5 - TESTES E ANÁLISE DE RESULTADOS

Para avaliar o desempenho das bibliotecas e do programa principal desenvolvidos neste trabalho, testes foram feitos e seus resultados foram analisados para demonstrar a eficiência da implementação e o funcionamento das técnicas empregadas.

5.1 Configuração de Hardware

Os testes foram realizados em um computador com a seguinte configuração de hardware:

- Notebook Dell Modelo D520;
- Processador: Intel(R) Core(TM)2 CPU T5500 @ 1.66GHz;
- Memória : 2GB DDR2;
- Adaptador de Vídeo: Mobile Intel(R) 945GM Express Chipset Family (256MB, PCI, PS 2.0);
- Disco Rígido: WDC WD1200BEVS-75UST0 120GB (SATA150, 2.5", 5400rpm, NCQ, 8MB Cache);

5.2 Configuração de Software

Os seguintes softwares foram utilizados para o desenvolvimento e os testes:

- Sistema Operacional: Microsoft Windows XP (2002) Professional 5.01.2600 (Service Pack 3);
- IDE de Desenvolvimento: Bloodshed Dev-C++, Versão 4.9.92;

5.3 Testes e Análises de Resultados

Os testes e as análises das bibliotecas e do programa principal foram feitos separadamente. Cada sub-seção a seguir mostra os resultados obtidos em cada uma dessas partes.

5.3.1 Biblioteca adaphuff

Os testes da implementação do algoritmo de compactação de Huffman adaptativo foram feitos com 15 arquivos, sendo 5 no formato .exe (executáveis) e 10 no .txt (texto), com tamanhos aproximados de 1 a 5 megabytes cada.

5.3.1.1 Teste 1.1 – Compactação de Arquivos .exe e .txt

Neste teste foi feita a compactação e a posterior descompactação de arquivos executáveis e de texto. Foram usados dois arquivos de texto diferentes: um contendo o alfabeto (26 caracteres) repetido várias vezes e outro com caracteres aleatórios. O objetivo do uso desses dois arquivos de texto é verificar como a frequência dos símbolos influencia na eficiência do algoritmo. O arquivo executável é usado para demonstrar que a adaptação da estrutura da árvore apresentada no Capítulo 4 funciona como esperado. O Gráfico 1 mostra o tempo de compactação necessário para cada arquivo e o Gráfico 2 mostra a eficiência de compactação.

Gráfico 1 - Tempo de compactação, em segundos, dos arquivos texto e executáveis.

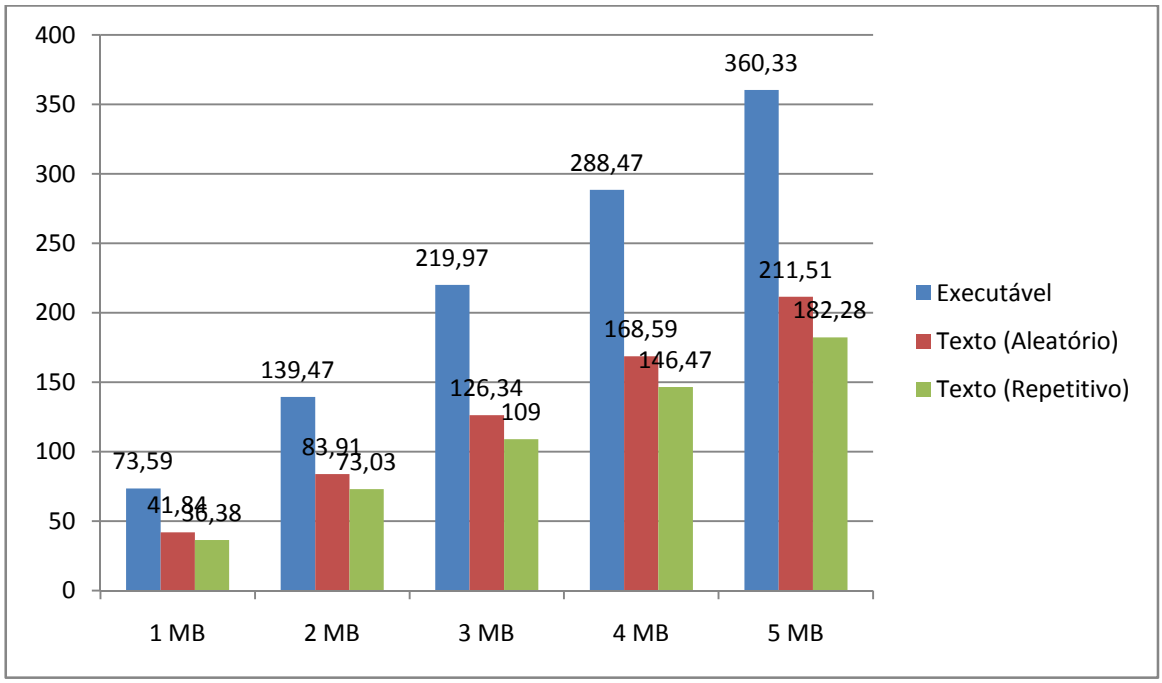
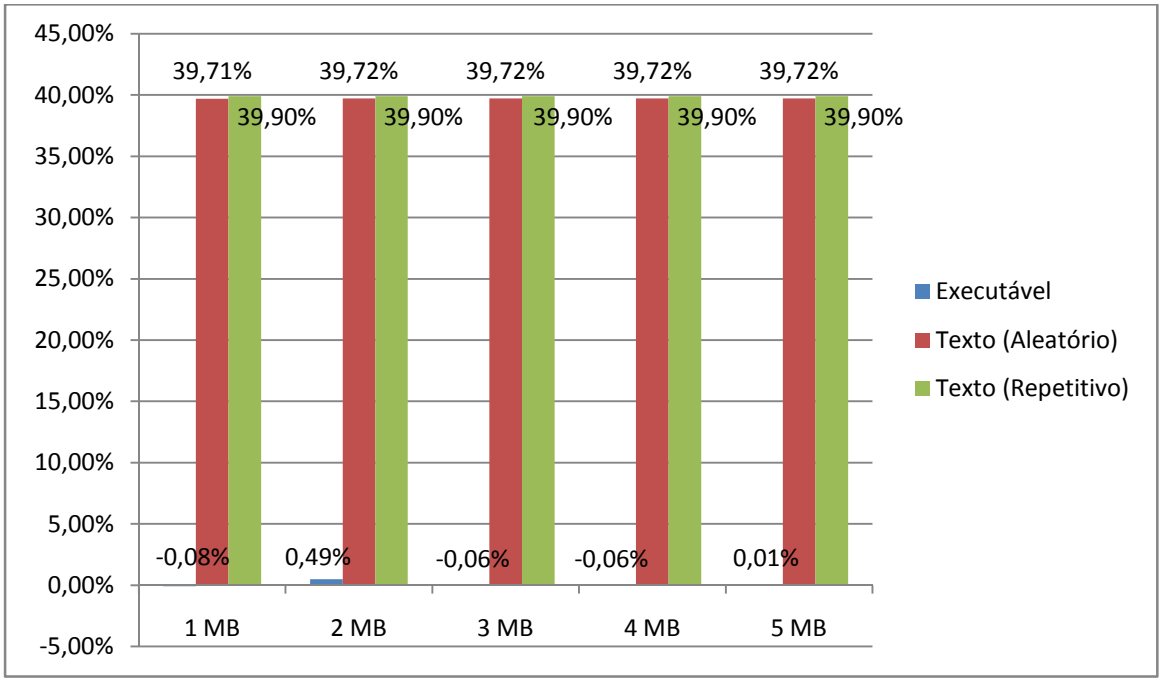


Gráfico 2 - Eficiência, em porcentagem, da compactação de arquivos texto e executáveis.

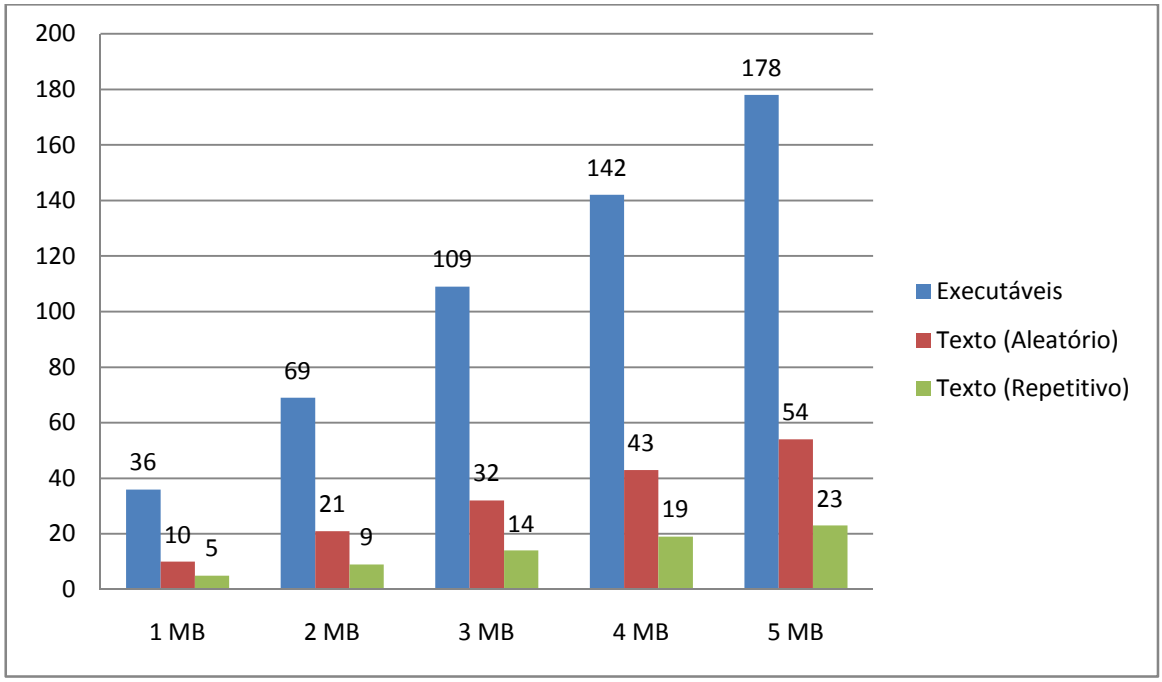


Nota-se pelo Gráfico 1 que tanto arquivos executáveis quanto de texto aleatório tomam mais tempo para serem compactados do que os textos repetitivos. Isso se deve ao fato de haver mais pesquisas por caracteres já existentes e percursos maiores feitos na árvore, para escrever os códigos de Huffman. Ao analisar o Gráfico 2 é percebida a grande diferença de eficiência entre a compactação de arquivos de texto e executáveis, chegando a ser negativa para os executáveis, ou seja, o arquivo de saída é maior que o executável original. Conclui-se que arquivos executáveis têm pouca redundância em sua estrutura, diferente de arquivos textos que têm várias repetições, como o espaço que é dado entre palavras.

5.3.1.2 Teste 1.2 – Descompactação de Arquivos .exe e .txt

Ao fazer a descompactação dos arquivos do Teste 1.1 e analisar o tempo de descompactação, temos os resultados mostrados no Gráfico 3.

Gráfico 3 - Tempo de descompactação, em segundos, dos arquivos texto e executáveis.



Por não precisar fazer buscas na árvore para saber ser o caractere já existe, a descompactação é bem mais eficiente em tempo de processamento. A proporção de tempo gasto entre os diferentes tipos de arquivos, em comparação com a compactação, foi mantida na descompactação.

É concluído então que quanto maior a quantidade de caracteres existentes em um arquivo e menor suas repetições, mais demorado e menos eficiente é o processo de compactação e descompactação. Para arquivos com poucos caracteres que são repetidos várias vezes, o algoritmo de Huffman é muito eficiente, tanto em tempo de processamento quanto na razão de compactação.

5.3.1.3 Teste 1.3 – Compactação e Descompactação de Imagens .bmp

Partindo das conclusões do teste anterior foram feitos testes com imagens bitmap com dominância de cores e com imagens com várias cores misturadas. Para isso foram escolhidas duas imagens, as Figuras 14 e 15.

Figura 14 - Imagem com várias cores e em vários tons diferentes.

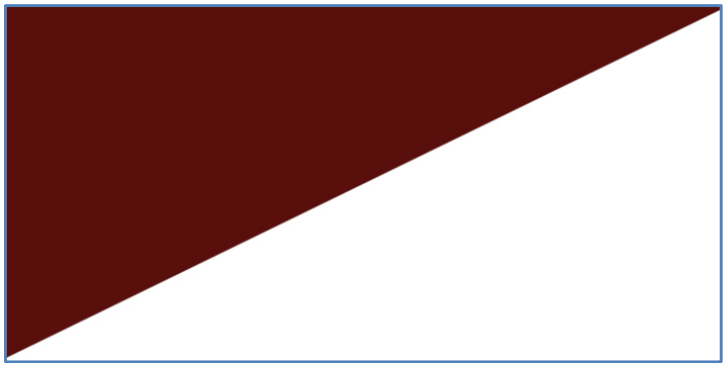


Figura 15 - Imagem bicromática.

Fazendo a compactação da Figura 14 tem-se uma redução de 6,41% no seu tamanho em 77.22 segundos, passando de 1.222.890 bytes a 1.144.530 bytes. Já com a Figura 15 foi obtida uma compactação de 71,12% em 29,13 segundos de 742.054 bytes para 214.320 bytes.

Para fazer a descompactação da Figura 14 são gastos 37,61 segundos contra apenas 7,08 segundos para a Figura 15. Portanto, conclue-se que a repetição de cores em imagens é tão vantajosa para a eficiência do algoritmo quanto repetições de letras em arquivos texto.

5.3.2 Biblioteca stglsb

Para os testes da biblioteca de esteganografia desenvolvida foram utilizados arquivos executáveis, de imagem, documento e áudio.

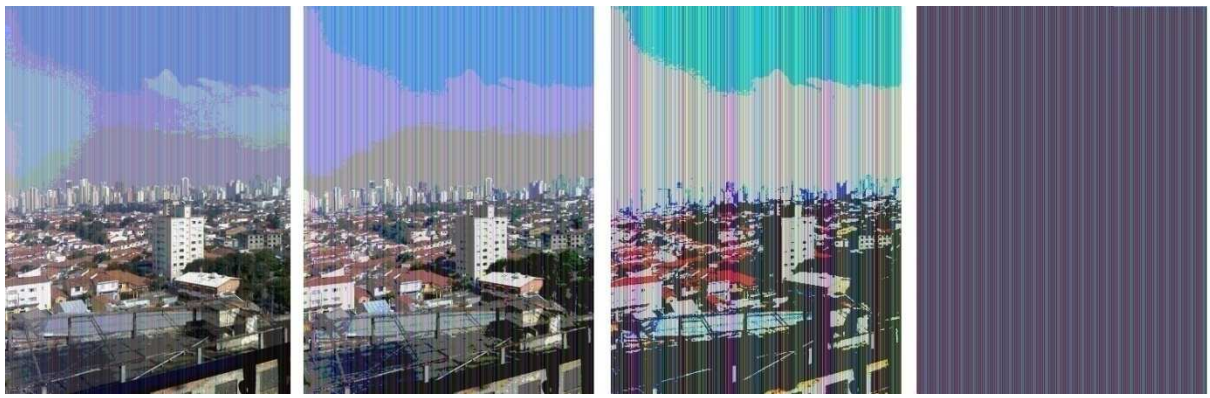
5.3.2.1 Teste 2.1 – Degradação Progressiva de Uma Imagem de Cobertura

A quantidade de dados que uma imagem consegue ocultar depende das dimensões dessa imagem e de quantos LSBs serão substituídos em cada pixel. Não se pode esquecer do princípio da visibiliade, ou seja, as alterações que dados ocultos podem causar a uma imagem de cobertura. Quantidades seguras de LSBs deve ser escolhidas para que não haja interceptação da imagem e destruição, ou mesmo extração, da mensagem oculta. Neste teste um arquivo é inserido várias vezes em uma imagem, fazendo uma substituição gradual de 1 a 8 LSBs e verificando os impactos visuais. A imagem de cobertura que foi usada em todos os testes desta biblioteca é a Figura 16, uma imagem bitmap de resolução 1200 por 1600 pixels e tamanho de arquivo de 5,49 MB. A quantidade total de bits que uma imagem pode ocultar é ((altura x largura) x 3) x *Qlsb*, sendo 3 a quantidade de bytes que representam um pixel e *Qlsb* a quantidade de LSBs a substituir. Nesta imagem teríamos 5.760.000 bits disponíveis com a substituição de apenas 1 LSB por byte. As Figuras 17 e 18 mostram os resultados da alterações de 1 a 8 LSBs de uma imagem bitmap, de dimensões 360 por 480 pixels e 518.454 bytes de tamanho

Figura 16 - Imagem de cobertura “sacada.bmp”.

(a) **(b)** **(c)** **(d)**

Figura 17 - Resultados das substituições de 1 a 4 bits menos significativos.



(a) **(b)** **(c)** **(d)**

Figura 18 - Resultados das substituições de 5 a 8 bits menos significativos.

Não há qualquer indício visual de alterações na primeira e na segunda imagens da Figura 15. Contudo, se olharmos seus histogramas e compará-los com o da imagem original, teremos a visão da Figura 19. São verificadas alterações anormais nas cores vermelho e verde, para 1 LSB e em todas as cores para 2 LSBs. Essa “sombra” que se nota no histograma leva à suspeita de alteração da imagem, mesmo que não se tenha a original à disposição, abrindo a possibildade de um ataque de extração ou de destruição da mensagem oculta.

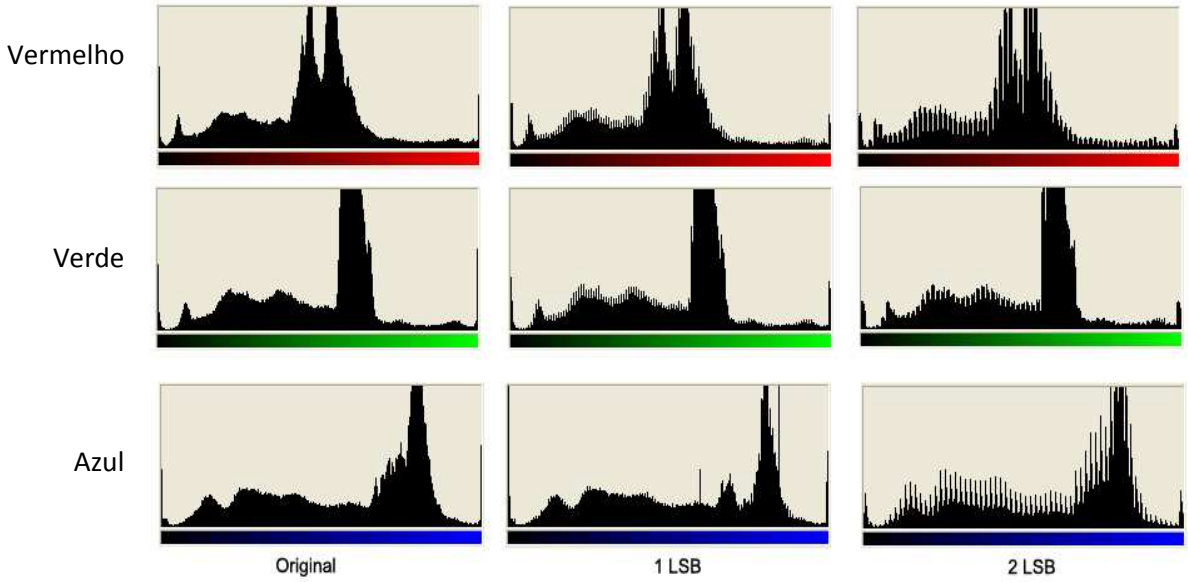


Figura 19 - Comparativo do histograma da imagem original com os de imagens que tiveram 1 e 2 LSBs alterados.

Esta análise nos leva à conclusão de que, apesar de ser uma técnica eficiente quanto a quantidade de armazenamento de dados, a esteganografia por LSB é frágil a ataques e facilmente identificável.

5.3.2.2 Teste 2.2 – Inserção e Extração de Arquivos

Para garantir que a implementação do algoritmo funciona corretamente, alguns testes simples de inserção e extração foram feitos. Foram usados nestes testes: um arquivo executável (.exe), um documento (.doc), uma imagem (.jpg) e um arquivo de áudio (.mp3). O arquivo bitmap usado de cobertura foi a Figura 16.

- **Arquivo executável**

Foi inserido na imagem um arquivo executável de 1.093.438 bytes, que ocupou 75,93% da imagem, com substituição de 2 LSBs, e consumiu 6,80 segundos. A extração demorou 5,73 segundos, recuperando os 1.093.438 bits de informação oculta. A execução do programa extraído ocorreu normalmente.

- **Arquivo de documento**

O arquivo .doc do Microsoft Word 2003 de 2.593.765 bytes levou 13,47 segundos para ser oculto, necessitando de 4 LSBs da imagem, ocupando 90,06% de sua área. Para recuperá-la foram necessários 12,70 segundos, extraindo corretamente o arquivo, sem falhas.

- **Arquivo de imagem**

Uma das possibilidades mais interessantes da esteganografia é a possibilidade de embutir imagens dentro de outras imagens. Foi feita a inserção de uma imagem .jpg de 228.953 bytes na imagem de cobertura com o uso de 1 LSB, ocupando 31,08% dos bytes da imagem em somente 2,98 segundos. A extração foi bem sucedida, levando 1.36 segundos.

- **Arquivo de áudio**

Para o teste com o arquivo de áudio foi utilizada uma música no formato .mp3 de 3.251.061 bytes de tamanho, de 2:15 minutos de duração. Com utilização de 5 LSBs foram ocupados 90,31% da imagem em 16,38 segundos. Sua extração durou 15,98 segundos e a música foi reproduzida sem falhas.

A inserção e a extração funcionam da mesma maneira para qualquer tipo de arquivo, pois suas estruturas internas são irrelevantes no processo. Os testes feitos concluem que a biblioteca desenvolvida é capaz de operar com qualquer tipo de arquivo e proceder sua recuperação integral, funcionando como esperado.

5.3.3 Programa Principal

Como os detalhes de desempenho de cada biblioteca já foram testados e validados, o que resta a analisar do programa principal é o funcionamento do conjunto e sua performance de tempo. Para isso serão usados três dos tipos de arquivos já utilizados nos testes anteriores: executável, imagem e áudio.

5.3.3.1 Teste 3.1 – Inserção e Extração de Arquivo Executável

Para o teste com esse formato foi utilizado o arquivo de 1 MB do Teste 1.1, inserindo-o na mesma imagem de cobertura utilizada até agora. O processo de inserção foi bem sucedido, consumindo 80,23 segundos e utilizando 2 LSBs de cada byte da imagem. A Figura 20 mostra a estego-imagem, visivelmente inalterada.

Figura 20 - Estego-imagem contendo um arquivo executável de 1MB compactado.

Para finalizar o teste foi feita a extração e o teste de funcionamento do arquivo. Levando 41,94 segundos pra extrair, o descompactador recriou o arquivo com sua extensão e tamanho originais. A Figura 21 mostra a instalação do software sendo executada a partir do arquivo recuperado. Assim, o sistema funciona para arquivos executáveis, mostrando o risco que um sistema de esteganografia pode representar se usado com propósitos criminosos, como por exemplo, disseminação de vírus.

5.3.3.2 Inserção e Extração de Arquivo de Imagem

O formato .jpg foi escolhido para este teste por seu tamanho já reduzido, usando poucos bytes da imagem de cobertura. A imagem original tem 228.953 bytes e resolução de 1600 por 1200 pixels. Por já ser um formato de imagem compactado, o arquivo compactado teve aumento de 0,07% do tamanho original. A inserção utilizou 1 LSB de cada byte da imagem, cobrindo um total de 31,82% da imagem de cobertura. Todo o processo durou 18,22 segundos. Na extração foram necessários somente 8,94 segundos, sendo a imagem perfeitamente extraída (Figura 21).

Figura 21 - Imagem "bar.jpg" extraída da estego-imagem.

Portanto, o sistema também é capaz de inserir e extrair imagens embutidas em arquivos bitmaps.

5.3.3.3 Inserção e Extração de Arquivo de Áudio

Descrito no Capítulo 1 em utilizações da esteganografia, o uso de arquivos de imagens inocentes para transmitir áudio foi feito por um traficante para dar ordens de execução de rivais a comparsas. Neste teste será demonstrada essa possibilidade. Tomamos um arquivo .mp3 de 2.125.824 bytes, o suficiente para armazenar 1:30 de áudio de ótima

qualidade. Após passados 150,86 segundos dos processos de compactação e inserção, o arquivo foi reduzido a 2.088.448 bytes, com compactação de 1,74% e usando 3 LSBs da imagem de cobertura. Em seguida foi feita sua extração e descompactação, que levou 79,44 segundos, recuperando o arquivo completamente audível, levando à conclusão que o uso da técnica pode ser utilizado também para áudio.

5.4 Conclusões Finais

Ao usar a compactação de dados o sistema desenvolvido faz algo semelhante à criptografia, que é dissimular os bits de suas sequências corretas, confundindo um possível atacante. Além disso, também há o ganho em espaço de armazenamento, principalmente com arquivos de texto que têm muita repetição de caracteres.

Foi demonstrado que o sistema pode inserir e extrair qualquer tipo de arquivo em imagens bitmap e fazer sua recuperação posterior sem erros ou falhas. Todos os objetivos propostos foram alcançados e os testes feitos demonstraram a eficiência de um sistema esteganográfico.

CAPÍTULO 6 - Conclusões e Trabalhos Futuros

Por fim, neste Capítulo é feita a conclusão do trabalho, expondo suas contribuições para a área, suas deficiências e fazendo sugestões de trabalhos futuros.

6.1 Conclusão

Após o estudo e a implementação com sucesso de um sistema de esteganografia LSB aliado ao algoritmo de Huffman adaptativo, é mostrada a capacidade da esteganografia em manter confidencialidade de dados sigilosos. Ela está presente à nossa volta, mesmo que não a reconheçamos pelo nome ou, como na maioria das vezes, não a vemos agir, já que esse é seu propósito. As aplicações sugeridas vão desde ocultação de mensagens secretas à marcação digital de direitos autorais invisível e detecção de adulterações eletrônicas, podendo no futuro tornar a técnica um sinônimo de segurança e sigilo, como é a criptografia hoje.

6.2 Contribuições

Espera-se que este trabalho sirva como incentivo para novos estudos e que seja para eles fonte de informações. As bibliotecas criadas também são contribuições que são deixadas para serem reutilizadas e melhoradas por outros desenvolvedores. Que trabalhos como este despertem a criatividade de outros estudantes e sirvam como incentivo para a criação de novas técnicas.

6.3 Limitações

Mesmo funcionando como previsto, a implementação dos algoritmos sofre de algumas deficiências, como a ineficiente busca linear pelo vetor auxiliar, no algoritmo de compactação de Huffman, e a utilização de imagens bitmap como imagem de cobertura, que apesar de suas qualidades pelas quais ela foi a escolhida para este trabalho, é um formato antigo e que é raramente usado.

6.4 Trabalhos Futuros

A partir das deficiências encontradas na implementação, são deixadas aqui sugestões para a melhoria do sistema e novos estudos sobre a esteganografia.

6.4.1 Técnica de Esteganografia Empregada

Foi visto que apesar de eficiente em armazenamento, a técnica de esteganografia por LSB é muito vulnerável a ataques e ruídos inerentes de linhas de transmissão. A implementação de uma técnica mais robusta é necessária para um sistema que tenha como objetivo garantir mais segurança aos dados ocultos e sua posterior recuperação completa.

6.4.2 Formato de Imagem de Cobertura

Pelas imagens bitmap não serem mais usadas, principalmente após o aparecimento do JPEG, seria mais interessante uma implementação da técnica que utilize imagens de formato comum na Internet, como o próprio JPEG e o TIFF ou ainda formatos de vídeo como AVI e MPEG.

6.4.3 Alteração da Implementação de Huffman Adaptativo

Ao desenvolver a biblioteca de Huffman adaptativo, percebeu-se que o algoritmo escreve o caminho até o nó NYT da árvore para sinalizar a inserção de um novo caractere. Fica como sugestão que seja modificada essa forma de sinalização. Ao encontrar um novo caractere, durante a compactação, é enviado o bit “1” e após ele, o caractere a ser inserido. No caso de um caractere já existente, seria enviado o bit “0” e em seguida o caminho até ele. Dessa maneira, reduz-se a quantidade de bits usados para representar o nó NYT, visto que ele sempre estará no nível mais baixo da árvore, consumindo sempre a maior quantidade de bits para ser sinalizado o envio de um novo caractere.

6.4.4 Implementação de Algoritmos de Busca e Ordenação de Vetor

Um dos limitadores de velocidade de compactação da implementação do algoritmo de Huffman adaptativo implementado é o vetor auxiliar. Durante as inserções nele, os nós são colocados em sequência. Dessa maneira, para chegar ao último novo caractere inserido, é necessário conferir posição a posição do vetor, tendo que passar por muitos nós internos, que não têm efeito nenhum durante a busca. Por isso, seria mais eficiente implementar uma ordenação no vetor que colocasse os nós folha nas primeiras posições.

Além da ordenação seria também interessante a implementação de um algoritmo de busca em vetor mais eficiente do que a busca linear. Outra forma de se implementar a busca é a utilização de uma função recursiva, de melhor desempenho que uma busca em vetor.

REFERÊNCIAS

ALIGHIERI, Dante; et al. **. Clássicos Jackson**. Rio de Janeiro: W. M. Jackson, 1957. 377p.

ANDERSON, Ross J.; PETITCOLAS, Fabien A. P. **On The Limits of Steganography**. IEEE Journal of Selected Areas in Communications: [S.l.], 1998. p 474-481.

ANDERSON, Ross J.; KUHN, Markus G.; PETITCOLAS, Fabien A. P. **Information Hiding – A Survey**. Proceedings of the IEEE, special issue on protection of multimedia content: [S.l.], 1999. p. 1062-1078.

ARRUDA, Diego V. O.; GOES, Rodrigo S. **Programa Para Compactação de Dados Usando Código de Huffman.** Goiânia: 2003, 66f. Universidade Federal de Goiás, Goiás, 2003.

BANVILLE, Lee. Case Study: Jack Kelley and USA Today. **Online NewsHour**, [S.l.] 10 dez. 2004. Disponível em: <http://www.pbs.org/newshour/media/media_ethics/casestudy_usatoday.php>. Acesso em: 08 abr. 2008.

BENTO, Ricardo Jorba; COELHO, Cristina Machado. **Ferramentas de esteganografia e seu uso na infowar**. Universidade Católica de Brasília, Brasília, DF.

BLELLOCH, Guy E. **Introduction to Data Compression**. Carnegie Mellon University, 2001. Disponível em: <http://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www/compression.pdf>. Acesso em: 24 jun. 2008.

BRUNORI JUNIOR, Roberto. **Uma Teoria de Primeira Ordem para Especificação e Análise de Protocolos de Criptografia**. 1999. 198 f. Mestrado em Ciência da Computação - Universidade Federal do Ceará, Fortaleza, Ceará, 1999.

CAMPOS, Arturo S. E. **Static Huffman**. [S.l], 1999. Disponível em: <http://www.arturocampos.com/ac_static_huffman.html>. Acesso em: 24 jun. 2008.

CHIARAMONTE, Rodolfo Barros; ORDONEZ, Edward David Moreno; PEREIRA, Fábio Decêncio. **Criptografia em software e hardware**. São Paulo: Novatec, 2005. 288 p.

COELLO, Carlos A. C.; LEÓN, Héctor H.. **Compresión de Bases de Datos**. Actas del VIII Simposio Internacional en Aplicaciones de Informática. Antofagasta, Chile. 21/25 nov. 1994. páginas 87-94, 1994.

LOW, Jonathan. **Adaptive Huffman Coding**. Duke University, 2000. Disponível em: < http://www.cs.duke.edu/csed/curious/compression/>. Acesso em: 24 jun. 2008.

COLE, Eric. **Hiding in plain sight: Steganography and the art of covert communication.** Indianapolis: Wiley Publishing, Inc., 2003.

KELLEY, Jack. Terror groups hide behind Web encryption. **The USA Today**, [S.l.], 02 maio 2001. Disponível em: <http://www.usatoday.com/tech/news/2001-02-05-binladen.htm>. Acesso em: 08 abr. 2008.

KATZENBEISSER, Stefan; PETITCOLAS, Fabien A. P. **Information Hiding Techniques for Steganography and Digital Watermark**. Norwood: Artech House, 2000.

OREBAUGH, Angela D. **Steganalysis: A Steganography Intrusion Detection System**. George Mason University, 2004.

PFITZMANN, Birgit. **Information Hiding Technology**. Cambridge, Inglaterra: Information Hiding Workshop, 1996.

ROCHA, Anderson Rezende. **Camaleão: Um Software Para Segurança Digital Utilizando Esteganografia**. Universidade Federal de Lavras: 2003, 108f. Lavras, Minas Gerais, 2003.

SAADE, Débora C. M. **Técnicas de Compactação e Compressão**. Universidade Federal Fluminense, 2008. Disponível em: <http://www.midiacom.uff.br/~debora/fsmm/pdf/parte3.pdf>. Acesso em: 24 jun. 2008.

TERRA. Abadia usava Hello Kitty para enviar ordens. **Terra Networks**, [S.l.] 08 abr. 2007. Disponível em: <http://noticias.terra.com.br/brasil/interna/0,,OI2666590-EI5030,00.html>. Acesso em: 05 abr. 2008.

_____. Traficante Juan Carlos Abadia é preso pela PF em SP. Terra Networks, [S.l.: s.n.]. Disponível em: <http://noticias.terra.com.br/retrospectiva2007/interna/0,,OI2121900-EI10676,00.html>. Acesso em: 08 abr. 2008.

ZMOGINSKI, Felipe. Abadía usou e-mail cifrado para traficar. INFO Online, 10 mar. 2008. Disponível em: <http://info.abril.com.br/aberto/infonews/032008/10032008-3.shl>. Acesso em: 08 abr. 2008.

ANEXO A - Programa principal - main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[])
{
	char arquivoCompactado[FILENAME_MAX] = "";
	char arquivoDescompactado[FILENAME_MAX] = "";
	clock_t t_inicio, t_fim;

	if(argc > 4 || argc < 3)
		puts("Quantidade de parametros invalida.");
	else
	{
		if(!strcmp(argv[1],"-o") || !strcmp(argv[1],"-O"))
		{
			if(argc == 4)
			{
				t_inicio = clock();
				strcpy(arquivoCompactado,(char*)compactar(argv[2]));
				ocultarLSB(arquivoCompactado, argv[3]);
				t_fim = clock();
			}
			else
				puts("Quantidade de Parâmetros inválida.");
		}
		else if(!strcmp(argv[1],"-e") || !strcmp(argv[1],"-E"))
		{
			if(argc == 3)
			{
				t_inicio = clock();
				strcpy(arquivoDescompactado,(char*)extrairLSB(argv[2]));
				descompactar(arquivoDescompactado);
				t_fim = clock();
			}
			else
				puts("Quantidade de Parâmetros inválida.");
		}

	printf("************************************************************\n");
		printf("Tempo total do processo: %.2f segundos.\n",
					(double)(t_fim - t_inicio) / CLOCKS_PER_SEC);

		}

		return 0;
	}
```

ANEXO B - Biblioteca adaphuff - arvoreHuffman.c

```
#include "arvoreHuffman.h"

//Função que cria um novo nó e retorna um ponteiro para ele.
no* criarNo()
{
	no *novoNo;

	//Aloca memória para o novo nó.
	novoNo = malloc(sizeof(no));

	//Se não foi possível alocar o novo nó, exibe mensagem de erro e sai do
	//programa.
	if(novoNo == NULL)
	{
		puts("\n\n");
		printf("Erro ao alocar memoria para um novo no.");
		puts("\n\n");

		exit(0);
	}

	//Inicializa os dados do nó.
	novoNo->caractere = 0;
	novoNo->peso = 0;
	novoNo->ordem = 0;

	//Inicializa os ponteiros do nó.
	novoNo->pai = NULL;
	novoNo->esquerdo = NULL;
	novoNo->direito = NULL;

	//Retorna o ponteiro criado.
	return novoNo;
}

//Função que cria uma nova árvore. Árvore deve ser passada como referência.
void criarArvore(AHA *novaAHA)
{
	no *novaRaiz;

	//Cria um novo nó que será a raiz da árvore.
	novaRaiz = criarNo();

	//Atribui ao novo nó a ordem máxima da árvore.
	novaRaiz->ordem = ORDEM_MAXIMA;

	//Transforma o nó raiz em nó NYT.
	novaRaiz->caractere = NYT;
```

```
	//Ponteiro para a raiz da árvore aponta para o novo nó criado.
	novaAHA->raiz = novaRaiz;

	//Inicializa o vetor auxiliar.
	inicializarVetor(novaAHA);

	//Inserir o nó raiz no vetor auxiliar.
	inserirEmVetor(novaAHA, novaRaiz);
}


//Função que insere um novo caractere na árvore, no nó NYT. Ao chamá-la deve-se
//saber previamente que o caractere não existe na árvore ainda.
no* inserirNaArvore(AHA *aha, unsigned short caractere)
{
	int posicao;
	no *novoNYT, *antigoNYT, *novoNo;

	novoNo = criarNo();
	novoNYT = criarNo();

	//Procurar o nó NYT.
	posicao = procurarEmVetor(*aha, NYT);

	//Se não encontrou o nó NYT, exibe mensagem de erro e sai do programa.
	if(aha->vetor[posicao]->caractere != NYT)
	{
		puts("Inconsistencia ao procurar o no NYT durante a insercao de novo
caractere.\n");
		exit(0);
	}

	//Aponta para o nó NYT, que será expandido.
	antigoNYT = aha->vetor[posicao];

	//Criação do novo nó, com o novo caractere.
	novoNo->caractere = caractere;
	novoNo->peso = 1;
	novoNo->ordem = antigoNYT->ordem - 1;

	//Criação do novo NYT.
	novoNYT->caractere = NYT;
	novoNYT->ordem = antigoNYT->ordem - 2;

	//Transforma o nó NYT antigo em nó interno
	antigoNYT->caractere = NO_INTERNO;
	antigoNYT->esquerdo = novoNYT;
	antigoNYT->direito = novoNo;
```

```
        //Atualizar os ponteiros de nós pais.
        novoNYT->pai = antigoNYT;
        novoNo->pai = antigoNYT;

        //Inserir no vetor auxiliar o novo nó NYT e o nó do novo caractere.
        inserirEmVetor(aha, novoNo);
        inserirEmVetor(aha, novoNYT);

        //Retorna o antigo nó NYT, já que os nós abaixo não serão mais manipulados.
        //Esse retorno facilita ao chamar a função atualizarArvore().
        return antigoNYT;
}

//Função que verifica se o nó passado como parâmetro deve ser trocado com outro
//nó.
no* maiorOrdemDaClasse(AHA *aha, no *pesquisado)
{
        no *tmp;
        int i;

        tmp = pesquisado;

        //Procura em todos os nós da árvore, pelo vetor.
        for(i=0; i<ORDEM_MAXIMA; i++)
        {
                //Se chegar ao final do vetor, o nó não precisa ser trocado.
                if(aha->vetor[i] != NULL)
                {
                        //Se o nó pesquisado estiver na mesma classe de peso, mas ordem
                        //menor do que o do nó apontado, deve fazer a troca.
                        if(aha->vetor[i]->peso == tmp->peso &&
                                aha->vetor[i]->ordem > tmp->ordem)
                        {
                                //Se não for pai do nó pesquisado e não for nó raiz, tmp
                                //recebe o nó a ser trocado.
                                if(aha->vetor[i] != pesquisado->pai &&
                                                        !ehRaiz(aha->vetor[i]))
                                        tmp = aha->vetor[i];
                        }
                }
                else break;
        }

        //Se não deve ser trocado, retorna NULL.
        if(tmp == pesquisado)
                return NULL;
        //Se deve, retorna o nó encontrado.
        else
                return tmp;
}
```

```
//Função que atualiza os pesos e faz trocas de nós da árvore.
void atualizarArvore(AHA *aha, no *noInicial)
{
	no *encontrado, *noAtual;

	//Recebe o nó a partir do qual a árvore será atualizada.
	noAtual = noInicial;

	//Enquanto não chega ao nó raiz, atualiza nó por nó.
	while(!ehRaiz(noAtual))
	{
		//Verifica se há necessidade de fazer troca de nós.
		encontrado = maiorOrdemDaClasse(aha, noAtual);

		//Se não retornou NULL deve trocar o noAtual com outro nó.
		if(encontrado != NULL)
		{
			no *no1, *no2, *paiNo1, *paiNo2;
			unsigned int ordem;

			//Atribui os ponteiros auxiliares.
			no1 = noAtual;
			no2 = encontrado;
			paiNo1 = no1->pai;
			paiNo2 = no2->pai;

			//Se forem filhos do mesmo pai.
			if(paiNo1 == paiNo2)
			{
				if(paiNo1->direito == no1)
				{
					paiNo1->direito = no2;
					paiNo1->esquerdo = no1;
				}
				else if(paiNo1->esquerdo == no1)
				{
					paiNo1->esquerdo = no2;
					paiNo1->direito = no1;
				}
			}
			//Se não forem filhos do mesmo pai.
			else
			{
				//Troca os nós filhos.
				if(paiNo1->direito == no1)
					paiNo1->direito = no2;

				if(paiNo1->esquerdo == no1)
					paiNo1->esquerdo = no2;
```

```
                        if(paiNo2->esquerdo == no2)
                                paiNo2->esquerdo = no1;

                        if(paiNo2->direito == no2)
                                paiNo2->direito = no1;
                }

                //Troca os nós pais.
                no1->pai = paiNo2;
                no2->pai = paiNo1;

                //Troca as ordens
                ordem = no1->ordem;
                no1->ordem = no2->ordem;
                no2->ordem = ordem;
        }

        //Atualiza o peso do nó.
        noAtual->peso++;

        //Vai para o nó pai.
        noAtual = noAtual->pai;
    }
}

//Função que escreve em arquivoTemp o caminho na árvore até o nó com o caractere
//passado como parâmetro.
void escreverCaminho(AHA aha, unsigned short caractere, FILE *arquivoTemp)
{
    int posicao, i;
    no *noAtual, *paiNoAtual;

    char codigo[12] = "";

    //Procura pela existência do caractere no vetor auxiliar.
    posicao = procurarEmVetor(aha, caractere);

    //Se o caractere não existe, procura o NYT e escreve o caminho até ele.
    if(aha.vetor[posicao] == NULL)
        posicao = procurarEmVetor(aha, NYT);

    //Define o nó de onde partirá o percurso.
    noAtual = aha.vetor[posicao];

    //Enquanto não chegar à raiz, continua percorrendo a árvore.
    while(!ehRaiz(noAtual))
    {
        //Armazena um ponteiro para o nó pai.
        paiNoAtual = noAtual->pai;
```

```
            if(noAtual == paiNoAtual->direito)
                strcat(codigo,"1");
            else
                strcat(codigo,"0");

            noAtual = noAtual->pai;
        }

        //Sendo o caminho percorrido de baixo para cima, é necessário fazer a
        //inversão da string de caminho.
        strrev(codigo);

        //Escreve a string no arquivo temporário.
        for(i=0; codigo[i] != '\0'; i++)
            fwrite(&codigo[i], sizeof(unsigned char), 1, arquivoTemp);

        //Depois de escrito o caminho, escreve o caractere em binário, se novo.

        //Aqui é desfeito o deslocamento antes de escrever, pois para escrever no
        //arquivo destino, os caracteres não podem ser maiores que 255(8 bits),
        //caso contrário seriam necessários mais dois bits (+3 dec) para escrever os
        //caracteres, o que diminuiria o desempenho da compactação.

        //NOTA: NYT, NO_INTERNO e PSEUDO_EOF não são escritos no arquivo
        //destino somente o caminho até o nó PSEUDO_EOF.

        //Se é um caractere novo, escreve os bits do caractere no arquivo.
        if(aha.vetor[posicao]->caractere == NYT)
        {
            int i;
            char byte[8];

            //Desfaz o deslocamento.
            caractere -= DESLOCAMENTO;

            //Grava os bits no vetor do bit menos pro mais significativo(invertido).
            for(i=7; i>=0; i--)
            {
                byte[i] = (caractere % 2) + 48;
                caractere /= 2;
            }

            //Escreve os bits como caracteres no arquivo temporário.
            for(i=0; i<8; i++)
                fwrite(&byte[i], sizeof(unsigned char), 1, arquivoTemp);
        }
}
```

```
//Função que retorna 1 se o nó é folha e 0 se é nó interno.
int ehFolha(no *tmp)
{
	//Se um dos nós filhos é NULL mas o outro não, há inconsistência na árvore.
	if((tmp->direito == NULL && tmp->esquerdo != NULL) ||
		(tmp->direito != NULL && tmp->esquerdo == NULL))
	{
		puts("Inconsistencia de no folha.");
		exit(0);
	}

	//Se ambos nós filhos forem NULL, é folha, então retorna 1.
	if(tmp->direito == NULL && tmp->esquerdo == NULL)
		return 1;
	//Se não for, retorna 0.
	else return 0;
}

//Função que retorna 1 se o nó é raiz e 0 se não é.
int ehRaiz(no *tmp)
{
	//Se o nó tiver ordem de nó raiz e o ponteiro pai for NULL, é raiz.
	if(tmp->ordem == ORDEM_MAXIMA)
	{
		if(tmp->pai == NULL)
			return 1;
		//Se o nó raiz não tem pai NULL, há inconsistencia na árvore.
		else
		{
			puts("Inconsistencia de ponteiro pai do no raiz.");
			exit(0);
		}
	}
	//Se não é raiz retorna 0;
	else
		return 0;
}

//Função que imprime os dados do nó passado como parâmetro.
void imprimirNo(no *tmp)
{
	if(tmp != NULL)
	{
		printf("%c(%d) ", tmp->caractere, tmp->caractere);
		printf("p = %d |", tmp->peso);
		printf("o = %d |\n", tmp->ordem);

		if(tmp->direito != NULL)
		{
			printf("\t Filho direito: ");
```

```
                printf("%c(%d) ", tmp->direito->caractere,
                                                tmp->direito->caractere);
                printf("p = %d |", tmp->direito->peso);
                printf("o = %d |\n", tmp->direito->ordem);

                printf("\t Filho esquerdo: ");
                printf("%c(%d) ", tmp->esquerdo->caractere,
                                                tmp->esquerdo->caractere);
                printf("p = %d |", tmp->esquerdo->peso);
                printf("o = %d |\n", tmp->esquerdo->ordem);
            }
            else
                puts("\t Sem Filhos.");
        }
}

//Função que conta e imprime a altura da árvore, sendo 0 a altura mínima.
void contarAltura(AHA aha)
{
        int posicao, niveis;
        no *noAtual;

        //Procura o no de insercao
        posicao = procurarEmVetor(aha, NYT);

        //Aponta para o no encontrado
        noAtual = aha.vetor[posicao];

        //Inicializa com a altura minima
        niveis = 0;

        //Enquanto nao chegar a raiz
        while(noAtual->pai != NULL)
        {
            //Incrementa o nivel
            niveis++;

            //Vai para o no pai
            noAtual = noAtual->pai;
        }

        printf("Altura da arvore: %d\n",niveis);
}

no* percorrerArvore(no* noAtual, unsigned char direcao)
{
        if(noAtual->direito != NULL)
        {
            if(direcao == 1)
                noAtual = noAtual->direito;
```

```
                else if(direcao == 0)
                        noAtual = noAtual->esquerdo;
        }
}



/********************** Funções de manipulação de vetor ************************/


//Função que inicializa o vetor auxiliar.
void inicializarVetor(AHA *aha)
{
        int i;

        //Atribui NULO a todas as posições do vetor.
        for(i=0; i<=ORDEM_MAXIMA; i++)
                aha->vetor[i] = NULL;
}

//Função que insere um novo ponteiro no vetor auxiliar.
void inserirEmVetor(AHA *aha, no *novoNo)
{
        int posicao;

        //Recebe a primeira posição vazia do vetor.
        posicao = procurarEmVetor(*aha, novoNo->caractere);

        //Posição do vetor aponta para o novo nó.
        if(aha->vetor[posicao] == NULL)
                aha->vetor[posicao] = novoNo;
        else
        {
                //Se a posição encontrada não for vazia.
                puts("Inconsistencia do vetor. Posicao de insercao nao encontrada.");
                exit(0);
        }

}

//se o caractere existir, retorna sua posicao no vetor
//senao retorna a primeira posicao vazia

//Função que faz a busca por caracteres no vetor. O deslocamento já deve ter
//sido aplicado ao caractere.
int procurarEmVetor(AHA aha, unsigned short caractere)
{
        int i;

        for(i=0; i<ORDEM_MAXIMA; i++)
```

```
	{
		//Se alcançou uma posição vazia ou encontrou o caractere, retorna a
		//posição do vetor.
		if(aha.vetor[i] == NULL || aha.vetor[i]->caractere == caractere)
			return i;
	}
}

//Função que imprime todos os nós do vetor, na ordem que estão inseridos.
void imprimirVetor(AHA aha)
{
	int i;

	for(i=0; i<ORDEM_MAXIMA; i++)
	{
		if(aha.vetor[i] != NULL)
		{
			printf("vetor[%d] = ",i);
			imprimirNo(aha.vetor[i]);
			puts("");
		}
		else
			break;
	}
}
```

ANEXO C - Biblioteca adaphuff – arvoreHuffman.h

```
#include <stdio.h>
#include <stdlib.h>

#define NYT 0
#define NO_INTERNO 1
#define PSEUDO_EOF 2
#define DESLOCAMENTO 3
#define ORDEM_MAXIMA 512 + DESLOCAMENTO

//Estrutura dos nós da árvore.
typedef struct no
{
	unsigned short caractere;
	unsigned int peso;
	unsigned int ordem;

	struct no *esquerdo;
	struct no *direito;
	struct no *pai;

} no;

//Estrutura da árvore de Huffman, com o vetor auxiliar e a raiz da árvore.
typedef struct AHA
{
	no *raiz;
	no *vetor[ORDEM_MAXIMA];
} AHA;

//Funções de manipulação da árvore.
no* criarNo();
void criarArvore(AHA *novaAHA);
no* inserirNaArvore(AHA *aha, unsigned short caractere);
no* maiorOrdemDaClasse(AHA *aha, no *pesquisado);
void atualizarArvore(AHA *aha, no *noInicial);
void escreverCaminho(AHA aha, unsigned short caractere, FILE *arquivoTemp);
void imprimirNo(no *tmp);
void contarAltura(AHA aha);
no* percorrerArvore(no* noAtual, unsigned char direcao);

//Funções de manipulação do vetor.
void inicializarVetor(AHA *aha);
void inserirEmVetor(AHA *aha, no *novoNo);
int procurarEmVetor(AHA aha, unsigned short caractere);
void imprimirVetor(AHA aha);
```

ANEXO D - Biblioteca adaphuff – compactador.c

```
#include "compactador.h"
#include "arvoreHuffman.h"

//Declarada fora da função para poder retornar seu nome ao final.
char nomeArquivoCompactado[FILENAME_MAX] = "";

char* compactar(char *nomeArquivoOrigem)
{
	//Variáveis para manipulação dos arquivos.
	FILE *arquivoTemp,
		 *arquivoOrigem,
		 *arquivoDestino;

	unsigned short caractere;
	unsigned char chtemp;

	int i, j;

	//Estrutura para geração de relatórios.
	struct rel
	{
		fpos_t f_inicio, f_fim;

		float tamanhoOriginal;
		float tamanhoCompactado;

		clock_t t_inicio, t_fim;
		clock_t tempoCompactacao;

		float porcentagemCompactacao;
	}rel;


	//Variáveis para extração da extensão do arquivo original.
	unsigned char extensao[3] = "", *ext, byte[8], ch;

	//Variáveis para manipulação da árvore.
	AHA arvore;
	int posicao;
	no *noAtual;

/********************** ABERTURA E CRIAÇÃO DOS ARQUIVOS
**********************/

	//Início da compactação.
	rel.t_inicio = clock();

	puts("\n\n");
```

```
printf("Compactando... ");

//Abre arquivo de origem dos dados como leitura.
arquivoOrigem = fopen(nomeArquivoOrigem, "rb");

if(arquivoOrigem == NULL)
{
        puts("\n\n");
        printf("Erro ao abrir %s para leitura.", nomeArquivoOrigem);
        puts("\n\n");

        exit(0);
}

//Calcula o tamanho total do arquivo de origem.
//Marca o início do arquivo.
fgetpos(arquivoOrigem,&rel.f_inicio);

//Vai até o fim do arquivo.
fseek(arquivoOrigem, 0, SEEK_END);

//Marca o fim do arquivo.
fgetpos(arquivoOrigem,&rel.f_fim);

//Retorna ao início.
fsetpos (arquivoOrigem,&rel.f_inicio);


//Cria o arquivo temporário.
arquivoTemp = tmpfile();

if(arquivoTemp == NULL)
{
        puts("\n\n");
        printf("Erro ao criar arquivo temporario.");
        puts("\n\n");

        exit(0);
}

//Extrai a extensão do arquivo original e a escreve no arquivo temporário.

//Procura a última ocorrência de "." para encontrar a extensão.
ext = strrchr(nomeArquivoOrigem,'.');

//Mudar a extensão do arquivo de origem.
strncpy(nomeArquivoCompactado, nomeArquivoOrigem,
                strlen(nomeArquivoOrigem)- strlen(ext));

strcat(nomeArquivoCompactado,"_compactado");
```

```
	strcat(nomeArquivoCompactado,".ahf");

	//Gravar a extensão no início do arquivo compactado.
	strncpy(extensao, ext+1 , strlen(ext));

	for(i=0; i<3; i++)
	{
		//Converte caractere por caractere.
		ch = extensao[i];

		//Decompõe o byte do caractere em bits.
		//Já atribui como decimal 0 ou 1 (+48)
		for(j=7; j>=0; j--)
		//for(j=0; j<8; j++)
		{
			byte[j] = (ch % 2) + 48;
			ch /= 2;
		}

		byte[8] = '\0';

		//Escreve os bits como caracteres no arquivo temporário.
		for(j=7; j>=0; j--)
			fwrite(&byte[j], sizeof(unsigned char), 1, arquivoTemp);
	}

	//Criar o arquivo onde os dados compactados serão finalmente escritos.
	arquivoDestino = fopen(nomeArquivoCompactado, "wb");

	if(arquivoDestino == NULL)
	{
		puts("\n\n");
		printf("Erro ao abrir %s para gravacao.", nomeArquivoCompactado);
		puts("\n\n");

		exit(0);
	}

/********************** COMPACTAÇÃO
*****************************************/

	//Cria árvore vazia.
	criarArvore(&arvore);

	//Inserir o PSEUDO_EOF.
	noAtual = (no*)inserirNaArvore(&arvore, PSEUDO_EOF);
	atualizarArvore(&arvore, noAtual);

	//Lê os dados até o final do arquivo temporário.
	while(!feof(arquivoOrigem))
```

```
{
	//Somente de 0 a 255
	fread(&chtemp,sizeof(unsigned char),1,arquivoOrigem);

	//Deslocamento para formatação à representacao interna.
	caractere = chtemp + DESLOCAMENTO;

	if(!feof(arquivoOrigem))
	{
		//Verifica a existência do caractere.
		posicao = procurarEmVetor(arvore, caractere);

		//Escreve caminho até o caractere, se ele já existir, ou até o NYT
		//se não existir.
		escreverCaminho(arvore, caractere, arquivoTemp);

		//Se for um caractere novo, insere-o na árvore.
		if(arvore.vetor[posicao] == NULL)
			noAtual = (no*)inserirNaArvore(&arvore, caractere);
		else
			noAtual = arvore.vetor[posicao];

		//Atualiza os pesos e faz trocas de nós, se necessário.
		atualizarArvore(&arvore, noAtual);
	}
};

//Escrever o caminho até o PSEUDO_EOF, sinalizando fim dos dados.
escreverCaminho(arvore, PSEUDO_EOF, arquivoTemp);

//Voltar ao início do arquivo temporário.
rewind(arquivoTemp);

//Converte, de 8 em 8, os 1´s e 0´s do arquivo temporário em byte.
while(!feof(arquivoTemp))
{
	int i;
	unsigned char chtmp;
	unsigned char caractere;

	caractere = 0;

	//Lê 8 "bits" e os transforma em um byte.
	for(i=7; i>=0; i--)
	{
		fread(&chtmp, sizeof(unsigned char), 1, arquivoTemp);

		if(!feof(arquivoTemp))
			caractere +=  pow(2,i)*(chtmp - 48);
	}
```

```
            //Escreve o byte obtido.
            fwrite(&caractere, sizeof(unsigned char), 1, arquivoDestino);
        };

        //Fim da compactação.
        rel.t_fim = clock();

        printf("Terminado!\n\n");

        //Fechar os arquivos utilizados
        fclose(arquivoOrigem);
        fclose(arquivoTemp);
        fclose(arquivoDestino);


/********************** RELATÓRIO
*****************************************/
        //Grava o tamanho do arquivo original.
        rel.tamanhoOriginal = rel.f_fim - rel.f_inicio;


        //Abrir o arquivo onde os dados compactados foram escritos.
        arquivoDestino = fopen(nomeArquivoCompactado, "rb");

        if(arquivoDestino == NULL)
        {
            puts("\n\n");
            printf("Erro ao abrir %s para leitura.", nomeArquivoCompactado);
            puts("\n\n");

            exit(0);
        }

        //Marca o início do arquivo.
        fgetpos (arquivoDestino,&rel.f_inicio);

        //Vai até o fim do arquivo.
        fseek(arquivoDestino, 0, SEEK_END);

        //Marca o fim do arquivo.
        fgetpos(arquivoDestino,&rel.f_fim);

        //Fecha o arquivo.
        fclose(arquivoDestino);


        //Grava o tamanho do arquivo compactado.
        rel.tamanhoCompactado = rel.f_fim - rel.f_inicio;
```

```
        //Grava o tempo gasto para compactação.
        rel.tempoCompactacao = rel.t_fim - rel.t_inicio;

        //Calcula a porcentagem de compactação.
        rel.porcentagemCompactacao = 100 -
                                                (rel.tamanhoCompactado * 100) /
rel.tamanhoOriginal;

        puts("****************** Compactacao ******************");

        printf("Tamanho original do arquivo: %.0f bytes.\n", rel.tamanhoOriginal);

        printf("Tamanho compactado: %.0f bytes.\n", rel.tamanhoCompactado);

        printf("Porcentagem de compactacao: %.2f%%.\n",
rel.porcentagemCompactacao);

        printf("Razao de compactacao: 1:%.2f.\n",

        rel.tamanhoOriginal/rel.tamanhoCompactado);

        printf("Tempo de compactacao: %.2f segundo(s).\n",
(double)rel.tempoCompactacao / CLOCKS_PER_SEC);

        puts("\n");

        return nomeArquivoCompactado;
}
```

ANEXO E - Biblioteca adaphuff – compactador.h

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

char* compactar(char *nomeArquivoOrigem);
```

ANEXO F - Biblioteca adaphuff – descompactador.c

```
#include "descompactador.h"
#include "arvoreHuffman.h"

void descompactar(char *nomeArquivoCompactado)
{
	//Variáveis para manipulação dos arquivos.
	FILE *arquivoCompactado,
		 *arquivoTemp,
		 *arquivoDestino;

	char nomeArquivoDestino[FILENAME_MAX] = "";
	int i, j;

	//Estrutura para geração de relatórios.
	struct rel
	{
		fpos_t f_inicio, f_fim;

		float tamanhoCompactado;
		float tamanhoOriginal;

		clock_t t_inicio, t_fim;
		clock_t tempoDescompactacao;

		float porcentagemCompactacao;
	}rel;

	//Variáveis para extração da extensão do arquivo compactado.
	unsigned char *ext, extensao[3] = "", byte, ch;

	//Variáveis para manipulação da árvore.
	AHA arvore;
	no* noAtual;


/***************ABERTURA E CRIAÇÃO DOS
ARQUIVOS*********************************/

	//Início da descompactação.
	rel.t_inicio = clock();

	puts("\n\n");
	printf("Descompactando... ");

	//Abre o arquivo compactado.
	arquivoCompactado = fopen(nomeArquivoCompactado, "rb");

	if(arquivoCompactado == NULL)
```

```
{
        puts("\n\n");
        printf("Erro ao abrir %s para leitura.", nomeArquivoCompactado);
        puts("\n\n");

        exit(0);
}

//Calcula o tamanho total do arquivo compactado.

//Marca o início do arquivo.
fgetpos(arquivoCompactado,&rel.f_inicio);

//Vai até o fim do arquivo.
fseek(arquivoCompactado, 0, SEEK_END);

//Marca o fim do arquivo.
fgetpos(arquivoCompactado,&rel.f_fim);

//Retorna ao início.
fsetpos (arquivoCompactado,&rel.f_inicio);


//Cria arquivo temporário.
arquivoTemp = tmpfile();

if(arquivoTemp == NULL)
{
        puts("\n\n");
        printf("Erro ao criar arquivo temporario.");
        puts("\n\n");

        exit(0);
}

//Recupera a extensão do arquivo compactado.
for(i=0; i<3; i++)
{
        //Lê 1 byte.
        fread(&byte, sizeof(unsigned char), 1, arquivoCompactado);

        ch = 0;

        for(j=7; j>=0; j--)
        {
                ch += pow(2,j)* (byte % 2);
                byte /= 2;
        }

        extensao[i] = ch;
```

```
	}

	//Insere o caractere de fim de string.
	extensao[3] = '\0';

	//Procura a última incidência de '.' .
	ext = strrchr(nomeArquivoCompactado,'.');

	//Muda a extensão do arquivo compactado.
	strncpy(nomeArquivoDestino, nomeArquivoCompactado,
			strlen(nomeArquivoCompactado)- strlen(ext));

	//Adiciona o sufixo "_compactado" ao nome do arquivo descompactado.
	strcat(nomeArquivoDestino,"_descompactado.");

	//Adiciona a extensão original do arquivo.
	strcat(nomeArquivoDestino,extensao);

	//Lê o arquivo compactado até o final e converte os bytes em bits (como
caracteres).
	while(!feof(arquivoCompactado))
	{
		unsigned char byteInverso;
		unsigned char byte[8];

		//Lê 1 byte para decompô-lo.
		fread(&byteInverso, sizeof(unsigned char), 1, arquivoCompactado);

		if(!feof(arquivoCompactado))
		{
			int i;

			//Decompõe o byte em bits.
			for(i=7; i>=0; i--)
			{
				byte[i] = byteInverso % 2;
				byteInverso /= 2;
			}

			//Grava os bits no arquivo temporário.
			for(i=0; i<8; i++)
				fwrite(&byte[i], sizeof(unsigned char), 1, arquivoTemp);
		}
	}

	//Fecha o arquivo compactado.
	fclose(arquivoCompactado);

	//Volta ao início do arquivo temporário.
	rewind(arquivoTemp);
```

```
        //Cria o arquivo de destino onde os dados descompactados serão restaurados.
        arquivoDestino = fopen(nomeArquivoDestino, "wb");
        if(arquivoDestino == NULL)
        {
                puts("\n\n");
                printf("Erro ao abrir %s para gravacao.", nomeArquivoDestino);
                puts("\n\n");

                exit(0);
        }

/************************** DESCOMPACTAÇÃO
************************************/

        //Cria uma árvore
        criarArvore(&arvore);

        //Insere o PSEUDO_EOF.
        noAtual = (no*)inserirNaArvore(&arvore, PSEUDO_EOF);
        atualizarArvore(&arvore, noAtual);

        //Lê os dados até o final do arquivo temporário.
        while(!feof(arquivoTemp))
        {
                int i;
                unsigned char bit;
                unsigned short caractere;

                //Se é um novo caractere.
                if(noAtual->caractere == NYT)
                {
                        caractere = 0;

                        //Recupera o caractere do arquivo.
                        for(i=7; i>=0; i--)
                        {
                                fread(&bit, sizeof(unsigned char), 1, arquivoTemp);
                                caractere += pow(2,i)*bit;
                        }

                        //Escreve o novo caractere no arquivo de destino.
                        fwrite(&caractere, sizeof(unsigned char), 1, arquivoDestino);


                        //Desloca o valor do caractere para inserir na árvore.
                        caractere += DESLOCAMENTO;

                        //Insere o novo caractere na árvore e a atualiza.
                        noAtual = (no*)inserirNaArvore(&arvore, caractere);
```

```
                atualizarArvore(&arvore, noAtual);

                //Retorna ao nó raiz
                noAtual = arvore.raiz;
            }
            //Se é um caractere já existente.
            else
            {
                //Lê bit a bit para percorrer a árvore, até chegar a um nó folha.
                do
                {
                    fread(&bit, sizeof(bit), 1, arquivoTemp);
                    noAtual = (no*)percorrerArvore(noAtual, bit);

                } while(!ehFolha(noAtual));

                //Se alcançou o nó PSEUDO_EOF, finaliza a descompactação.
                if(noAtual->caractere == PSEUDO_EOF)
                    break;

                //Desfaz o deslocamendo do caractere.
                caractere = noAtual->caractere - DESLOCAMENTO;

                //Se não é um novo nó.
                if(noAtual->caractere != NYT)
                {
                    //Escreve o caractere no arquivo de destino.
                    fwrite(&caractere, sizeof(unsigned char), 1,
arquivoDestino);

                    //Atualiza a árvore, incrementando o peso do novo nó e
fazendo
                    //sua atualização
                    atualizarArvore(&arvore, noAtual);
                }
                //Se é um novo nó volta para o início, inserindo-o.
                else continue;

                //Retorna ao nó raiz.
                noAtual = arvore.raiz;
            }

        };

        printf("Terminado!\n\n");

        //Fecha o arquivos temporário.
        fclose(arquivoTemp);

        //Fim da descompactação.
```

```
        rel.t_fim = clock();

/********************** RELATÓRIO
****************************************/

        //Grava o tamanho do arquivo compactado.
        rel.tamanhoCompactado = rel.f_fim - rel.f_inicio;

        //Retorna ao início do arquivo de destino para calcular seu tamanho.
        rewind(arquivoDestino);

        //Marca o início do arquivo.
        fgetpos(arquivoDestino,&rel.f_inicio);

        //Vai até o fim do arquivo.
        fseek(arquivoDestino, 0, SEEK_END);

        //Marca o fim do arquivo.
        fgetpos(arquivoDestino,&rel.f_fim);

        //Fecha o arquivo.
        fclose(arquivoDestino);

        //Grava o tamanho do arquivo descompactado.
        rel.tamanhoOriginal = rel.f_fim - rel.f_inicio;

        //Grava o tempo gasto para compactação.
        rel.tempoDescompactacao = rel.t_fim - rel.t_inicio;

        //Calcula a porcentagem de compactação.
        rel.porcentagemCompactacao = 100 -
                                                (rel.tamanhoCompactado * 100) /
rel.tamanhoOriginal;

        puts("****************** Descompactacao ******************");

        printf("Tamanho do arquivo compactado: %.0f
bytes.\n",rel.tamanhoCompactado);

        printf("Tamanho original do arquivo: %.0f bytes.\n",rel.tamanhoOriginal);

        printf("Porcentagem de compactacao: %.2f%%.\n",
rel.porcentagemCompactacao);

        printf("Razao de compactacao: 1:%.2f.\n",

        rel.tamanhoOriginal/rel.tamanhoCompactado);

        printf("Tempo de descompactacao: %.2lf segundo(s).\n",
```

```
                                                (double) rel.tempoDescompactacao /
CLOCKS_PER_SEC);

        puts("\n");
}
```

ANEXO G - Biblioteca adaphuff – descompactador.h

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void descompactar(char *nomeArquivoCompactado);
```

ANEXO H - Biblioteca stglsb – stglsb.c

```
#include "stglsb.h"

//Função que insere um arquivo qualquer em uma imagem bitmap.
void ocultarLSB(char *nomeArquivoOrigem, char *nomeImagemOrigem)
{
	//Variáveis para manipulação de arquivos.
	FILE *arquivoOrigem,
		 *imagemOrigem,
		 *imagemDestino,
		 *arquivoTemp;

	char nomeImagemDestino[FILENAME_MAX] = "";
	unsigned char byteImg, byteTemp;
	int i, j, bitTemp, intTemp;

	//Variáveis para extração da extensão do arquivo original.
	unsigned char extensao[3] = "", *ext;

	//Estrutura para armazenamento e cálculo de informações.
	struct rel
	{
		clock_t t_inicio, t_fim;
		clock_t tempoOcultacao;

		fpos_t f_inicio, f_fim;

		//Quantidade de bits existentes na origem dos dados.
		unsigned int qtdeBitsOrigem;

		//Quantidade de bytes existentes na imagem original.
		unsigned int qtdeBytesImagem;

		//Armazena a quantidade total de bits que poderão ser substituídos na
		//área de imagem inteira.
		unsigned int qtdeBitsImagem;

		//Quantidade de LSBs que serão substituídos em cada byte de imagem.
		unsigned int qtdeLSB;

		//Variáveis para definir quantos bytes serão necessários para ocultar
		//o cabeçalho.
		unsigned int cabExtensao;
		unsigned int cabTamanhoOrigem;
		unsigned int cabTotal;

	}rel;
```

```
/*****************************************************************************
***/

        //Inicia contagem do tempo de ocultação.
        rel.t_inicio = clock();

        puts("\n\n");
        printf("Inserindo... ");

        //Abre o arquivo de origem dos dados.
        arquivoOrigem = fopen(nomeArquivoOrigem, "rb");
        if(arquivoOrigem == NULL)
        {
                puts("\n\n");
                printf("Erro ao abrir %s para leitura.", nomeArquivoOrigem);
                puts("\n\n");

                exit(0);
        }

//Calcula quantos bits a ocultar existem no arquivo de origem.
        //Grava as posições de início e fim dos dados de origem.
        fgetpos(arquivoOrigem, &rel.f_inicio);
        fseek(arquivoOrigem, 0, SEEK_END);
        fgetpos(arquivoOrigem, &rel.f_fim);

        ///Retorna ao início do arquivo.
        fsetpos(arquivoOrigem, &rel.f_inicio);

        //Grava quantos bits de dados existem no arquivo de origem.
        rel.qtdeBitsOrigem = (rel.f_fim - rel.f_inicio) * 8;


//Adiciona o sufixo "_estego" ao nome da imagem com os dados ocultos e extrai a
//extensão.

        //Procura a última ocorrência de "." para encontrar a extensão.
        ext = strrchr(nomeImagemOrigem,'.');

        //Se não for um bitmap, mostra um aviso e  finaliza o programa.
        if(strcmp(".bmp",ext))
        {
                puts("\n\n");
                printf("ERRO! Imagem informada não é bitmap (.bmp).");
                puts("\n\n");

                exit(0);
        }
```

```
        strncpy(nomeImagemDestino,                    nomeImagemOrigem,
strlen(nomeImagemOrigem)-

                                strlen(ext));
        strcat(nomeImagemDestino,"_estego.bmp");

        //Abre o arquivo de imagem de origem.
        imagemOrigem = fopen(nomeImagemOrigem,"rb");
        if(imagemOrigem == NULL)
        {
                puts("\n\n");
                printf("Erro ao abrir %s para leitura.", nomeImagemOrigem);
                puts("\n\n");

                exit(0);
        }

        //Cria o arquivo de imagem de destino.
        imagemDestino = fopen(nomeImagemDestino,"wb");
        if(imagemDestino == NULL)
        {
                puts("\n\n");
                printf("Erro ao criar %s.", nomeImagemDestino);
                puts("\n\n");

                exit(0);
        }

        //Copia cabeçalho de imagem.
        for(i=0; i<54; i++)
        {
                fread(&byteImg, sizeof(unsigned char), 1, imagemOrigem);
                fwrite(&byteImg, sizeof(unsigned char), 1, imagemDestino);
        }

//Calcula quanto bytes de área de imagem existem.
        //Grava as posições de início e fim dos dados de imagem.
        fgetpos(imagemOrigem, &rel.f_inicio);
        fseek(imagemOrigem, 0, SEEK_END);
        fgetpos(imagemOrigem, &rel.f_fim);

        //Retorna ao início da área de dados de imagem.
        fsetpos(imagemOrigem, &rel.f_inicio);

        //Grava quantos bytes existem na área de imagem.
        rel.qtdeBytesImagem = rel.f_fim - rel.f_inicio;


        //Se a imagem não suportar o mínimo de bytes necessários para ocultar 1 byte
        //de dado, exibe um erro e finaliza o programa.
```

```
        // qtdelsb máxima + extensao + quantidade dados + 1 byte
        //Quantidade mínima: 3 + 3 + 4 + 1 = 11 bytes
        if(rel.qtdeBytesImagem < 11)
        {
                puts("\n\n");
                printf("ERRO! Imagem menor que o tamanho minimo possivel para ");
                printf("realizar a ocultacao. Utilize uma imagem maior.");
                puts("\n\n");

                exit(0);
        }

//Calcula a quantidade de bits menos significativos que serão necessários para
//ocultar os dados.

        //Inicializa a quantidade de LSBs com 1;
        rel.qtdeLSB = 1;

        //Define valores para uso de 1 LSB.
        rel.cabExtensao = 24;
        rel.cabTamanhoOrigem = 32;
        rel.qtdeBitsImagem = rel.qtdeBytesImagem;

        // 3 bits para quantidade de bits a substituir +
        // bits necessários para armazenar a extensão (3 * 8 = 24)+
        // bit necessários para armazenar a quantidade de dados ocultos
        //(32 bits = 1 int).
        rel.cabTotal = 3 + rel.cabExtensao + rel.cabTamanhoOrigem;

        //Se a quantidade final de dados (cabecalho + dados) for maior que a
        //quantidade de bits em área de imagem disponível.
        while( (rel.cabTotal + rel.qtdeBitsOrigem) > rel.qtdeBitsImagem )
        {
                //Aumenta a quantidade de LSBs a substituir.
                rel.qtdeLSB++;

                //Atualiza a quantidade de bytes que cada parte necessitará.
                rel.qtdeBitsImagem = rel.qtdeBytesImagem * rel.qtdeLSB;

                rel.cabTotal = 3 +
                                (rel.cabExtensao + rel.cabTamanhoOrigem)/rel.qtdeLSB;

                //Se sobrarem bits soma um byte.
                if( ((rel.cabExtensao + rel.cabTamanhoOrigem)%rel.qtdeLSB) != 0)
                        rel.cabTotal += 1;
        }

        //Se 8 bits ainda não forem suficientes para comportar os dados, exibe um
        //aviso e encerra o programa.
        if(rel.qtdeLSB > 8)
```

```
	{
		puts("\n\n");

		printf("A imagem nao comporta a quantidade de dados do arquivo ");
		printf("informado. \nUtilize um arquivo menor ou uma imagem maior.\n");

		puts("\n\n");

		exit(0);
	}

	//Após serem definidos todos os parâmetros dos dados a ocultar, faz a
	//criação de um arquivo temporário, com os dados "binarizados".

	//Cria o arquivo temporário.
	arquivoTemp = tmpfile();
	if(arquivoTemp == NULL)
	{
		puts("\n\n");
		printf("Erro ao criar arquivo temporario.");
		puts("\n\n");

		exit(0);
	}

	//Escreve a quantidade de LSBs direto no arquivo de imagem destino.
	//Somente serão necessários 3 bits, usando sempre 1 LSB.
	/*
		0|001 = 1
		0|010 = 2
		0|011 = 3
		0|100 = 4
		0|101 = 5
		0|110 = 6
		0|111 = 7
		1|000 = 8
	*/

	byteTemp = rel.qtdeLSB;

	for(i=0; i<3; i++)
	{
		fread(&byteImg, sizeof(unsigned char), 1, imagemOrigem);

		bitTemp = byteTemp % 2;
		byteTemp /= 2;

		byteImg >>= 1;
		byteImg <<= 1;
```

```
        byteImg += bitTemp;

        fwrite(&byteImg, sizeof(unsigned char), 1, imagemDestino);

}

//Escreve a extensão do arquivo oculto no arquivo temporário, em bits.
ext = strrchr(nomeArquivoOrigem,'.');
for(i=0; i<3; i++)
{
        byteTemp = ext[i+1];

        for(j=0; j<8; j++)
        {
                bitTemp = byteTemp % 2;
                byteTemp /= 2;

                fwrite(&bitTemp, sizeof(unsigned char), 1, arquivoTemp);
        }
}

//Escreve a quantidade de bits que serão lidos na extração.
intTemp = rel.qtdeBitsOrigem;
for(i=0; i<32; )
{
        bitTemp = intTemp % 2;
        intTemp /= 2;

        i++;

        fwrite(&bitTemp, sizeof(unsigned char), 1, arquivoTemp);
}

//Se houver resto da divisão do tamanho total do arquivo pela quantidade de
//LSB substituída, complementa o restante dos bits não substituídos do byte
//com 0. Sempre completa todos os bits para não colocar em um mesmo byte
//bits de cabeçalho e bits de dados do arquivo de origem.
bitTemp = 0;
if(56%rel.qtdeLSB != 0)
        //Acrescenta a diferença entre a quantidade de LSBs que falta e quantos
        //são substituídos.
        for(i=0; i< rel.qtdeLSB - (56%rel.qtdeLSB); i++)
                fwrite(&bitTemp, sizeof(unsigned char), 1, arquivoTemp);

//Escreve todos os bits do arquivo de origem no arquivo temporário.
while(!feof(arquivoOrigem))
{
        fread(&byteTemp, sizeof(unsigned char), 1, arquivoOrigem);
```

```
            if(!feof(arquivoOrigem))
            {
                  for(i=0; i<8; i++)
                  {
                        bitTemp = byteTemp % 2;
                        byteTemp /= 2;

                        fwrite(&bitTemp, sizeof(unsigned char), 1, arquivoTemp);
                  }
            }
      }

      //Fecha o arquivo de origem de dados.
      fclose(arquivoOrigem);

/******************                                                    OCULTAÇÃO
**************************************************/

      //Retorna ao início do arquivo temporário.
      rewind(arquivoTemp);

      //Substitui os LSBs da imagem original e grava na imagem de destino.
      while(!feof(arquivoTemp))
      {
            bitTemp = 0;

            for(i=0; i<rel.qtdeLSB; i++)
            {
                  fread(&byteTemp, sizeof(unsigned char), 1, arquivoTemp);
                  bitTemp += pow(2,i) * byteTemp;
            }

            //Lê 1 byte da imagem de origem.
            fread(&byteImg, sizeof(unsigned char), 1, imagemOrigem);

            //Substitui os LSBs.
            byteImg >>= rel.qtdeLSB;
            byteImg <<= rel.qtdeLSB;
            byteImg += bitTemp;

            //Grava o novo byte na imagem de destino.
            fwrite(&byteImg, sizeof(unsigned char), 1, imagemDestino);
      }

      //Se terminou de inserir e ainda houver dados de imagem, copia o restante.
      while(!feof(imagemOrigem))
      {
            fread(&byteTemp, sizeof(unsigned char), 1, imagemOrigem);

            if(!feof(imagemOrigem))
```

```
                    fwrite(&byteTemp, sizeof(unsigned char), 1, imagemDestino);
        }

        //Fecha os arquivos utilizados
        fclose(arquivoTemp);
        fclose(imagemOrigem);
        fclose(imagemDestino);

        printf("Terminado!\n\n");

        //Encerra contagem do tempo de ocultação.
        rel.t_fim = clock();
/******************                                                    RELATÓRIO
**************************************************/

puts("****************** Ocultacao **********************");
printf("LSBs substituidos de cada byte: %d bit(s)\n",rel.qtdeLSB);

printf("Bits existentes na origem dos dados: %d bits.\n",

                    rel.qtdeBitsOrigem);

printf("Bytes de area de imagem original: %d bytes.\n",

                rel.qtdeBytesImagem);

printf("Bits na imagem original para substituicao com %d LSB: ", rel.qtdeLSB);
printf("%d bits.\n", rel.qtdeBitsImagem);

puts("\n");
printf("Cabecalho:\n");

printf(" Extensao: %d bits.\n", rel.cabExtensao);
printf(" Tamanho da origem: %d bits.\n", rel.cabTamanhoOrigem);
printf(" Tamanho total ocupado: %d bytes.\n", rel.cabTotal);

puts("");
printf("Total de bits inseridos: %d bits.\n",rel.cabTotal + rel.qtdeBitsOrigem);

puts("");

rel.tempoOcultacao = rel.t_fim - rel.t_inicio;
printf("Tempo de ocultacao: %.2f segundo(s).\n", (double)rel.tempoOcultacao /

                            CLOCKS_PER_SEC);


printf("Utilizacao dos bits da imagem: %.2f%%.\n",
                                (((float)rel.cabTotal + (float)rel.qtdeBitsOrigem) *
100 ) /
```

```
        (float)rel.qtdeBitsImagem);

puts("\n");
}


/*************************************************************************
***/

char nomeArquivoDestino[FILENAME_MAX] = "";

//Função que tenta extrair de um arquivo bitmap um arquivo oculto.
char* extrairLSB(char *nomeImagemOrigem)
{
        //Variáveis para manipulação de arquivos.
        FILE *imagemOrigem,
                 *arquivoTemp,
                 *arquivoDestino;

        char *ext;
        unsigned char byteImg, byteTemp;
        int i, j, bitTemp, temp;

        //Estrutura para armazenar e calcular informações da extração.
        struct rel
        {
                //Variáveis para calcular o tempo de extração.
                clock_t t_inicio, t_fim;
                clock_t tempoOcultacao;

                //Variáveis para calcular tamanho de arquivos.
                fpos_t f_inicio, f_fim;

                //Quantidade de bits existentes na origem dos dados.
                unsigned int qtdeBitsOrigem;

                //Quantidade de LSBs que serão substituídos em cada byte de imagem.
                unsigned int qtdeLSB;

                //Variáveis para recuperar os dados de cabeçalho.
                unsigned char cabExtensao[3];
                unsigned int cabTamanhoOrigem;

        }rel;

/*************************************************************************
***/
```

```
//Inicia contagem do tempo de extração.
rel.t_inicio = clock();

puts("\n\n");
printf("Extraindo... ");

//Abre o arquivo de imagem de origem.
imagemOrigem = fopen(nomeImagemOrigem,"rb");
if(imagemOrigem == NULL)
{
	puts("\n\n");
	printf("Erro ao abrir %s para leitura.", nomeImagemOrigem);
	puts("\n\n");

	exit(0);
}

//Ignorar cabeçalho (54 bytes).
fseek(imagemOrigem, 54, SEEK_SET);

//Cria o arquivo temporário.
arquivoTemp = tmpfile();
if(arquivoTemp == NULL)
{
	puts("\n\n");
	printf("Erro ao criar arquivo temporario.");
	puts("\n\n");

	exit(0);
}

//Extrair a quantidade de LSBs usados em cada byte.
rel.qtdeLSB = 0;
for(i=0; i<3; i++)
{
	fread(&byteTemp, sizeof(unsigned char), 1, imagemOrigem);
	rel.qtdeLSB += pow(2,i) * (byteTemp % 2);
}

//Se a quantidade de LSB for 0 converte para 8.
if(rel.qtdeLSB == 0) rel.qtdeLSB = 8;

//Extrair o estego-cabeçalho dos LSBs para o arquivo temporário.
// extensao + quantidade de bits ocultos
// 24 + 32 = 56
rel.cabTamanhoOrigem = 0;
for(i=0; i<56;)
{
	fread(&byteTemp, sizeof(unsigned char), 1, imagemOrigem);
```

```
            for(j=0; j<rel.qtdeLSB && i<56; j++)
            {
                  bitTemp = byteTemp % 2;
                  byteTemp /= 2;

                  i++;

                  fwrite(&bitTemp, sizeof(unsigned char), 1, arquivoTemp);
            }
      }

      //Volta ao início do arquivo termporário.
      rewind(arquivoTemp);

      //Extrair a extensão original.
      for(i=0; i<3; i++)
      {
            bitTemp = 0;

            //Lê 8 bits e transforma em um byte.
            for(j=0; j<8; j++)
            {
                  fread(&byteTemp, sizeof(unsigned char), 1, arquivoTemp);
                  bitTemp += pow(2,j) * byteTemp;
            }

            rel.cabExtensao[i] = bitTemp;
      }

      //Insere o caractere terminador de string.
      rel.cabExtensao[i] = '\0';


      //Extrair a quantidade de bits a ler.
      bitTemp = 0;
      for(i=0; i<32; i++)
      {
            fread(&byteTemp, sizeof(unsigned char), 1, arquivoTemp);
            bitTemp += pow(2,i) * byteTemp;
      }

      //Atribui a quantidade de bits à variável de relatório.
      rel.cabTamanhoOrigem = bitTemp;

//Copia todos os LSBs ocultos para o arquivo temporário.

      //Fecha o arquivo temporário.
      fclose(arquivoTemp);

      //Cria um novo arquivo temporário.
```

```
        arquivoTemp = tmpfile();
        if(arquivoTemp == NULL)
        {
                puts("\n\n");
                printf("Erro ao criar arquivo temporario.");
                puts("\n\n");

                exit(0);
        }

        //Atribui à variável temporária a quantidade total de bits a ler.
        temp = rel.cabTamanhoOrigem;

        //Gravar os LSBs de dados no arquivo temporário.
        while(!feof(imagemOrigem) && temp > 0)
        {
                //Lê 1 byte de imagem.
                fread(&byteImg, sizeof(unsigned char), 1, imagemOrigem);

                //Enquanto não chegar ao fim do arquivo, continua extraindo.
                if(!feof(imagemOrigem))
                {
                        //Extrai os LSBs utilizados.
                        for(i=0; i<rel.qtdeLSB && temp >0; i++)
                        {
                                bitTemp = byteImg % 2;
                                byteImg /= 2;

                                //Decrementa a quantidade de bits a ler.
                                temp--;

                                //Escreve o bit lido no arquivo temporário.
                                fwrite(&bitTemp, sizeof(unsigned char), 1, arquivoTemp);
                        }
                }
        }

//Criar o novo arquivo, com a extensão original do arquivo oculto.
        //Mudar a extensão do arquivo de origem.
        ext = strrchr(nomeImagemOrigem,'.');
        strncpy(nomeArquivoDestino,                    nomeImagemOrigem,
strlen(nomeImagemOrigem)-

                                strlen(ext));
        strcat(nomeArquivoDestino,"_extraido.");
        strcat(nomeArquivoDestino,rel.cabExtensao);

        //Cria o arquivo de destino original.
        arquivoDestino = fopen(nomeArquivoDestino,"wb");
        if(arquivoDestino == NULL)
```

```
        {
                puts("\n\n");
                printf("Erro ao criar %s.", nomeArquivoDestino);
                puts("\n\n");

                exit(0);
        }

        //Volta ao início dos dados.
        rewind(arquivoTemp);

        //Grava conjuntos de 8 bits em 1 byte para recuperar as informações.
        while(!feof(arquivoTemp))
        {
                byteTemp = 0;

                for(i=0; i<8; i++)
                {
                        fread(&bitTemp, sizeof(unsigned char), 1, arquivoTemp);
                        byteTemp += pow(2,i) * bitTemp;
                }

                fwrite(&byteTemp, sizeof(unsigned char), 1, arquivoDestino);
        }

        //Fecha os arquivos utilizados.
        fclose(arquivoTemp);
        fclose(imagemOrigem);
        fclose(arquivoDestino);

        printf("Terminado!\n\n");

        //Encerra a contagem do tempo de extração.
        rel.t_fim = clock();

/******************                                                    RELATÓRIO
**************************************************/

puts("****************** Extracao **********************");
printf("LSBs substituidos de cada byte: %d bit(s)\n",rel.qtdeLSB);

printf("Quantidade de dados extraidos: %d bits.\n",rel.cabTamanhoOrigem);

rel.tempoOcultacao = rel.t_fim - rel.t_inicio;
printf("Tempo de ocultacao: %.2f segundo(s).\n", (double)rel.tempoOcultacao /

                                CLOCKS_PER_SEC);


puts("\n");
```

```
return nomeArquivoDestino;
}
```

ANEXO I - Biblioteca stglsb – stglsb.h

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```