7
Huffman Homofônico-Canônico

7.1
Introdução

Neste capítulo será apresentado o algoritmo HHC - Huffman Homofônico-Canônico. Este algoritmo cria uma nova árvore homofônica baseada na árvore de Huffman canônica original para o texto de entrada. Os resultados de alguns experimentos são mostrados.

Este trabalho foi publicado no *Wseg - Workshop on Computer Systems Security*, Brazil, 2001.

7.2
Esquema geral de funcionamento

O foco deste algoritmo é a comercialização de grandes coleções textuais. O processo é o seguinte:

1. O cliente adquire a coleção (em CD, por exemplo);
2. Após a realização do pagamento, o cliente recebe uma chave secreta, através de um meio seguro. A chave secreta é que permite o acesso à coleção.

Na figura 7.1, encontra-se um esquema geral de funcionamento do algoritmo *HHC - Huffman Homofônico Canônico*. Na codificação, utilizando o *Arquivo Original*, é feita uma varredura (*parsing*) para determinar as freqüências de cada símbolo. Logo após, são criados os símbolos homofônicos para os símbolos originais. O próximo passo é determinar os códigos canônicos de cada símbolo homofônico. É também nesta fase que é produzido o *Arquivo Modelo*, utilizado no processo de decodificação. A seguir, utilizando a chave secreta do destinatário, é feita a permutação inicial dos símbolos homofônicos em cada nível da árvore. Como última etapa, tem-se a codificação do *Arquivo Original*, gerando o *Arquivo Cifrado* que é enviado ao destinatário, através de um canal inseguro.

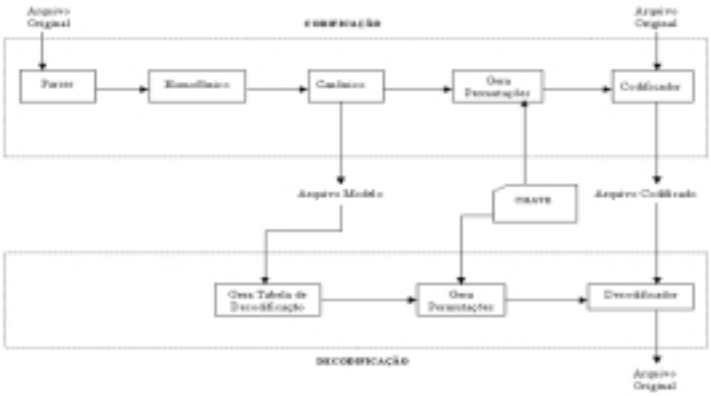

Figura 7.1: Visão Geral do Algoritmo *HHC - Huffman Homofônico Canônico.*

Na decodificação, utilizando as informações do *Arquivo Modelo*, é criada a tabela de decodificação. A configuração inicial da tabela de decodificação é baseada na permutação da chave secreta. Nesta etapa, também é feita a recriação dos símbolos homofônicos. Na última etapa, com o *Arquivo Cifrado*, é realizada a decodificação das informações.

A comparação do algoritmo *HHC* com o *Huffman Canônico* tem por objetivo mostrar que a compressão dos dados e o tempo de execução não ficam significativamente comprometidos.

7.3 O Processo de Codificação do algoritmo HHC

Este algoritmo é baseado no uso dos símbolos homofônicos e a permutação destes mesmos símbolos nos níveis da árvore, como técnicas básicas de difusão e confusão (30).

Devido às características do processo de codificação, o algoritmo *HHC* está organizado de forma modular. Na figura 7.2, estão representados todos os módulos do algoritmo, identificando a seqüência de execução, bem como os resultados de cada fase. A seguir, cada módulo é detalhado.

7.3.1 O Módulo Parser

Este módulo faz a varredura do texto para acumular as freqüências observadas, de cada símbolo do alfabeto no texto em claro. Os símbolos são armazenados em uma árvore binária de busca.

O algoritmo pode ser configurado para percorrer n-gramas ou palavras como símbolos. Os separadores são também considerados símbolos.

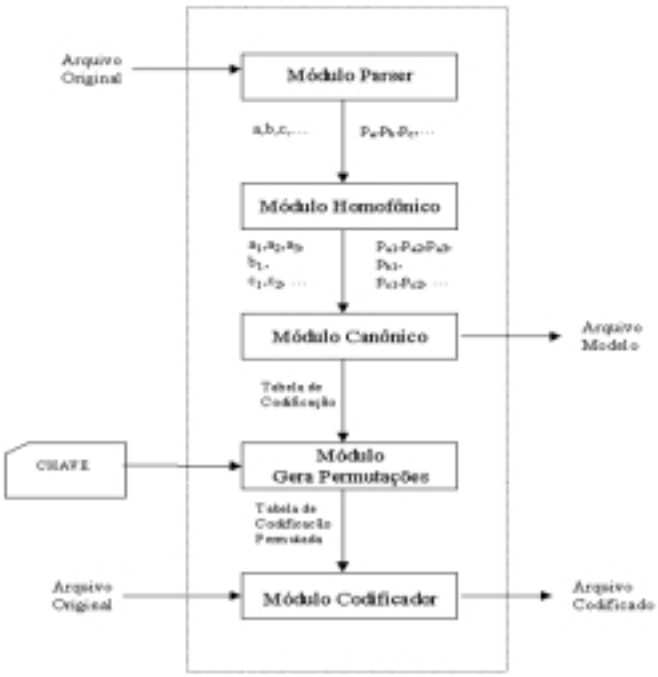

Figura 7.2: Processo de Codificação do *HHC*.

Nos experimentos, foi utilizado o *parsing* de palavras porque este apresenta taxas de compressão superiores, quando se trata de grandes coleções de texto.

7.3.2 O Módulo Homofônico

Neste módulo, foi utilizada a substituição homofônica de tamanho variável. Os homofônicos de cada símbolo original são representados pelos termos da decomposição em potências negativas de 2 do valor da sua freqüência relativa. Assim, é obtida uma distribuição diádica, ou seja, todas as freqüências relativas são potências negativas de 2.

Por exemplo, para o símbolo com probabilidade igual a 3/4, tem-se dois símbolos homofônicos: um com freqüência relativa igual a 2^{-1} e o outro igual a 2^{-2}. Como conseqüência deste fato, tem-se que o número total de símbolos pode crescer muito, tanto quanto o número de termos da decomposição. Outro fenômeno possível é a ocorrência de dízimas periódicas, implicando em um número infinito de homofônicos. Uma solução simples é o truncamento, eliminando as potências de menor valor. Por exemplo, neste algoritmo o número máximo de termos é limitado a 8. Mais à frente é justificada a escolha deste limite.

O procedimento abaixo faz a decomposição das probabilidades dos

símbolos originais em potências negativas de 2, criando, assim, os símbolos homofônicos:

```
void fatora_homo(P_LEXICAL node) {
   int i=0,n=0;
   double  p, v;
   P_LISTA_HOMO newnode;
   P_LISTA_HOMO node_lista = node->lista_homo;
   p = node->prob;
   v = 1;
   while (p != 0 && n < NUMERO_HOMOFONICOS) {
      i++;
      p = p * 2;
      v = v / 2.0;
      if (p >= 1) {
         p--;
         n++;
         newnode = (P_LISTA_HOMO)malloc(sizeof(T_LISTA_HOMO));
         if (node_lista == NULL)
            node->lista_homo = newnode;
         else
            node_lista->prox = newnode;
         newnode->indice = n;
         newnode->nivel = i;
         newnode->valor = v;
         newnode->prox = NULL;
         node_lista = newnode;
      }
   }
}
```

7.3.3 O Módulo Canônico

Com o objetivo de obter uma decodificação rápida, é utilizado o algoritmo de codificação via Códigos de Huffman Canônicos para determinar os códigos de cada símbolo homofônico. Este algoritmo necessita apenas do tamanho do código para realizar o processamento.

O caminho natural seria utilizar o algoritmo de Huffman tradicional para determinar os tamanhos dos códigos ótimos para os símbolos homofônicos.

Mas, como cada um dos símbolos possui probabilidade igual a uma potência negativa de 2 e a soma de todas as probabilidades é igual a 1, estes valores podem ser determinados analiticamente. Assim, um símbolo com probabilidade 2^{-l} terá tamanho de código igual a l. Então, o algoritmo *Homofônico Canônico* utiliza essa propriedade para gerar diretamente os códigos canônicos, mesmo considerando que a soma de todas as probabilidades pode não ser igual a 1, devido ao truncamento na decomposição.

Suponha que determinado texto é composto somente por três símbolos (A, B e C), e que todos tenham a mesma probabilidade, ou seja, a probabilidade de cada um é 1/3. Então, a decomposição em potências negativas de 2 é:

$\frac{1}{3} = -22 + -42 + -62 + ...$

Na figura 7.3, encontra-se a árvore canônica que representa os símbolos homofônicos para esta decomposição, truncada em 3 símbolos homofônicos para cada símbolo original.

Figura 7.3: Árvore canônica para 3 símbolos homofônicos de probabilidades iguais.

Observando o último nível da árvore canônica na figura 7.3, observa-se os 3 símbolos homofônicos (A_3, B_3 e C_3) e um nó pontilhado rotulado de F. Este último nó, não faz parte da decomposição e sinaliza a falta de um nó externo neste nível. Este fato não é previsto nos códigos de Huffman, já que o algoritmo de Huffman cria uma árvore cujos nós pais têm sempre dois nós filhos.

Devido a esta anomalia, ocorrida devido ao truncamento na decomposição, é necessário alterar o algoritmo de codificação dos códigos de Huffman Canônicos, proposto por Moffat et al. (22). A alteração é bastante simples. No item 3 do algoritmo está introduzido um arredondamento para cima no cálculo do *firstcode[i]*. O pedaço de código abaixo apresenta este item com a devida alteração.

```
3.firstcode[maxlength] = 0
  para l = 1 até maxlength - 1 passo -1 faça
      firstcode[l] = ceiling( (firstcode[l + 1] + num[l + 1]) / 2 )
```

No momento da codificação, quando um símbolo X é codificado, na verdade um de seus homofônicos $\{X_1, X_3, ..., X_8\}$ é o escolhido. Há duas alternativas simples para a escolha: seleção seqüencial, onde na primeira vez é usado o X_1, na segunda o X_2, até que após o X_8 inicia-se novamente em X_1 e assim por diante; ou a seleção aleatória (11), onde um dos homofônicos é sorteado independentemente da escolha anterior. Nos experimentos foi adotada a seleção aleatória.

7.3.4 O Módulo Gera Permutações

Este módulo utiliza uma chave secreta de tamanho variável (256/512/1024 bits, etc.) para a cifragem na codificação. Essa chave pode ser informada pelo usuário ou gerada automaticamente pela aplicação de nível superior.

Essa chave define permutações sobre os símbolos que pertencem a um mesmo nível da árvore canônica. Assim, é feita uma permutação inicial sobre os símbolos de mesmo tamanho de código como ilustrado na figura 7.4.

Supondo a existência de um alfabeto de símbolos {A, B, C, D, E, F} e os conjuntos de homofônicos definidos por decomposição das probabilidades como sendo os conjuntos $\{A_1, A_2\}$, $\{B_1, B_2, B_3\}$, $\{C_1, C_2, C_3\}$, $\{D_1, D_2\}$, $\{E_1, E_2, E_3, E_4, E_5, E_6, E_7, E_8\}$e $\{F_1, F_2, F_3, F_4, F_5\}$, respectivamente. Então, supondo que A_1, B_3, C_2, D_1, E_3 e F_4 possuam a mesma probabilidade igual a 2^{-x}, eles ocuparam o mesmo nível na árvore canônica

A cifragem consiste na permutação inicial destes símbolos de mesmo nível a partir da ordem definida pela chave secreta agrupada em bytes. Para uma chave de 1024 bits, tem-se até $\frac{1024}{8} = 128$ elementos a permutar, o que possibilita 128! permutações.

Por exemplo, suponha que uma chave de 1024 bits seja utilizada com valor $K = 00101000100000000010110...$, e que, agrupando-se K em bytes, tem-se os seguintes valores decimais: $K = 20.64.11.45.2.89.....$ Nesse caso, o vetor permutação é $P = (20, 64, 11, 45, 2, 89, ...)$, o que permite a permutação de um bloco de até 128 elementos.

Algumas considerações:

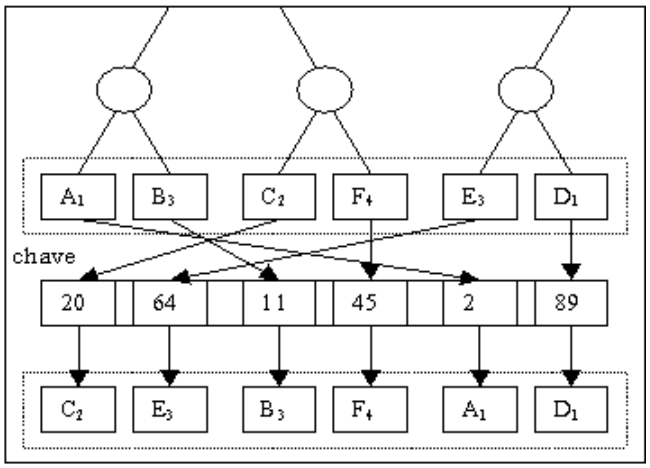

Figura 7.4: Permutação no nível da árvore.

– No caso de valores repetidos dentro do vetor permutação P, alguma lógica de tratamento de colisões deverá ser definida. Por exemplo, procurar a próxima posição livre pela fórmula $p' = (p+1) mod 128$.

– Caso um nível possua mais do que 128 símbolos, basta agrupá-los em vários blocos de 128 bits e aplicar as permutações a cada bloco. Abordagens alternativas para o caso de níveis com mais de 128 símbolos são: ciclar a chave, ou expandir a chave a partir de funções pseudo-aleatórias.

– Caso o nível possua menos que 128 símbolos, o vetor de permutação deve ser normalizado para permitir a permutação. Por exemplo, caso o vetor P possua 128 elementos como no exemplo anterior e o nível possua somente 6 símbolos, apenas os primeiros 6 valores de P são considerados, e a ordem entre esses valores define o novo vetor normalizado. Exemplo: $P = (20, 64, 11, 45, 2, 89, ...)$ gera $P' = (3, 5, 2, 4, 1, 6)$. Nesse caso, a permutação do bloco $(A_1, B_3, C_2, F_4, E_3, D_1)$ resulta em $(C_2, E_3, B_3, F_4, A_1, D_1)$ conforme mostrado na figura 7.4.

7.3.5 O Módulo Codificador

Este módulo faz a codificação do texto original, utilizando o sorteio dos símbolos homofônicos e a permutação dos níveis. Utilizando a figura 7.4, suponha que o símbolo A seja codificado. Para isto, é feito o sorteio dos homofônicos de A, conforme procedimento definido por Massey et al. (11). Se o símbolo A_1 for o escolhido, então o código que será gravado no arquivo codificado será o do símbolo C_2. O símbolo B_3 é substituído pelo E_3 e assim por diante. O que esse esquema propõe é que quando o símbolo A

for codificado, e o seu homofônico A_1 for o escolhido, na verdade o C_2 é que é codificado. Essa permutação amplia a homofonia uma vez que um símbolo representante (homofônico) de A, por exemplo, agora passa a ser codificado por um representante (homofônico) de C.

Opcionalmente, uma operação *Troca* (swap) pode ser aplicada fazendo com que o símbolo usado na codificação seja realocado para a primeira posição no nível (operação move-to-front). Por exemplo, quando o símbolo B_3 é codificado, ele fará com que o símbolo E_3 troque de lugar com o símbolo que ocupa a primeira posição. Assim, a cada codificação tem-se uma nova configuração e cada símbolo tem diferentes codificações a cada iteração, devido à escolha do homofônico que o representa ou devido à operação *Troca*.

7.4 O Processo de Decodificação do HHC

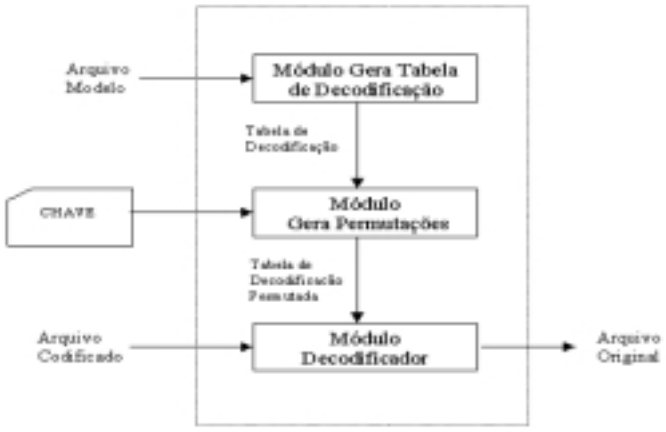

Figura 7.5: Processo de Decodificação.

O procedimento de decodificação utiliza o algoritmo correspondente dos códigos de Huffman Canônicos, associado à chave secreta do usuário para recriar as permutações. Na figura 7.5, encontram-se os módulos que compõem este procedimento.

7.4.1 O Módulo Gera Tabela de Decodificação

Este módulo, utilizando o *Arquivo Modelo*, tem a função de gerar a tabela de decodificação, como definida no algoritmo dos códigos de Huffman Canônicos. Ele também gera os símbolos homofônicos para cada símbolo do alfabeto. Esta nova decomposição da probabilidade em potências negativas

de 2, é necessária porque o *Arquivo Modelo*, para não prejudicar a taxa de compressão, só contém os símbolos originais com suas respectivas freqüências.

7.4.2
O Módulo Gera Permutações

Este módulo tem a mesma função de seu homônimo no procedimento de codificação, ou seja, ele define a configuração inicial de cada nível da árvore canônica.

7.4.3
O Módulo Decodificador

Este módulo utiliza o algoritmo de decodificação dos códigos de Huffman Canônicos, proposto por Moffat et al. (22). Ele também é responsável por desfazer as permutações, produzidas na codificação.

7.5
O Esquema de Utilização do Algoritmo

Nesta seção, são apresentadas as formas de utilização dos módulos do algoritmo *HHC*. A escolha do esquema mais adequado fica condicionada aos recursos disponíveis e ao grau de segurança desejado.

Com o objetivo de minimizar o custo computacional no momento da distribuição, o algoritmo *HHC* pode ser sempre executado em duas etapas. Para um determinado documento, primeiro podem ser executados os três primeiros módulos (*Parser*, *Codificador* e *Canônico*), e a estrutura resultante é armazenada. No ato da distribuição, de posse da chave secreta, é realizado o processamento dos módulos *Gera Permutações* e *Canônico*.

7.5.1
Tradicional

Na figura 7.6, o esquema de codificação *Tradicional* é mostrado. A principal vantagem deste esquema é o tempo de codificação. Duas desvantagens: tamanho do modelo, que é maior do que nos próximos esquemas; e a transmissão do modelo em aberto, isto é, o modelo não criptografado é transmitido em um canal inseguro.

7.5.2
Modelo Codificado Vulnerável

Uma forma de diminuir o tamanho do modelo é aplicar o próprio algoritmo ao *Arquivo Modelo*. Neste caso, a codificação do modelo é feita

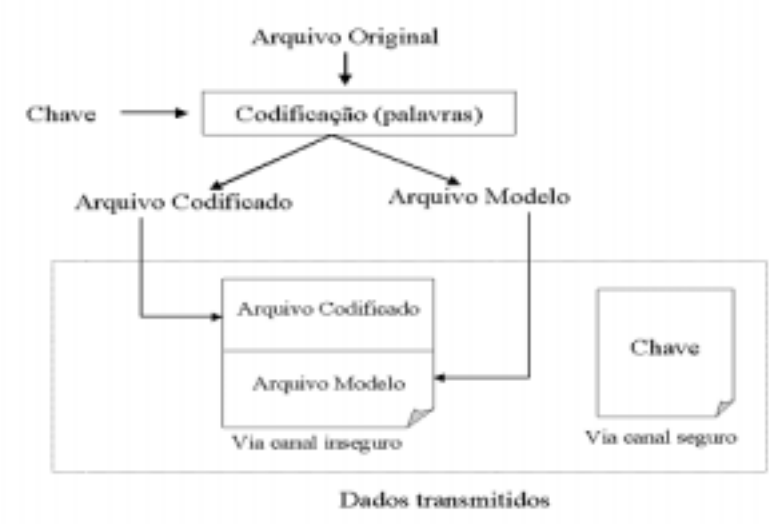

Figura 7.6: Esquema de Codificação *Tradicional*.

usando-se o *parsing* de caracteres, assim o novo modelo resultante tende a ser pequeno. A compressão do modelo poderia ser otimizada utilizando-se tratamentos específicos dado o formato particular do arquivo.

A figura 7.7 ilustra este esquema de codificação. Na Fase 1, é feita a codificação *Tradicional*, com a geração do *Arquivo Codificado* e do *Arquivo Modelo*. Na Fase 2, é feita a codificação do *Arquivo Modelo* gerando o *Modelo Codificado* e o novo *Arquivo Modelo*, utilizando o *parsing* de caracteres.

A vantagem deste esquema é aumentar o grau de dificuldade para o criptoanalista, devido a cifragem do modelo antigo e a criação de um novo. As desvantagens são o aumento do tempo de codificação - em relação ao esquema *Tradicional*, e o novo modelo ainda é transmitido em aberto.

7.5.3 Modelo Codificado Protegido

No esquema anterior foi feita a codificação do modelo, mas ainda persiste o problema da transmissão do modelo em aberto no canal inseguro, ou seja, o criptoanalista pode ter acesso as informações contidas no novo modelo. Aqui, este problema é solucionado.

Como o novo modelo é pequeno, devido a utilização do *parsing* de caracteres, e considerando que a chave é transmitida ao usuário através de um canal seguro, também é transmitido o novo modelo utilizando o mesmo canal seguro.

Na figura 7.8, encontra-se a disposição das informações transmitidas para o usuário, destacando aquelas transmitidas em canal inseguro e as que utilizam canal seguro. A principal vantagem deste esquema é o fato de esconder o

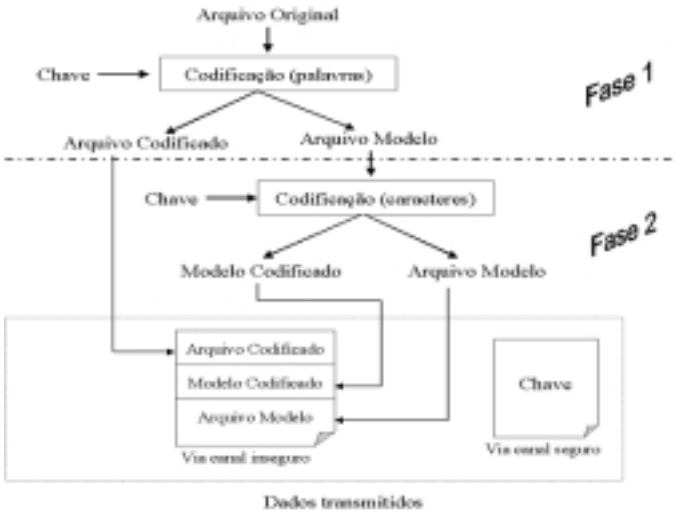

Figura 7.7: Esquema de Codificação *Modelo Codificado Vulnerável*.

modelo do criptoanalista. A desvantagem é ter que utilizar um canal seguro para a transmissão do modelo, apesar deste ser relativamente pequeno.

Figura 7.8: Esquema de Codificação *Modelo Codificado Protegido*.

7.5.4 Chave Composta com o Modelo

Este esquema utiliza a característica do algoritmo que permite a execução dos módulos separados. O processo de codificação é separado em dois blocos: no bloco chamado *Codificação* são executados os módulos *Parser*, *Homofônico* e *Canônico*; e no bloco *Cifragem* são executados os módulos *Gera Permutação* e *Codificar*.

A figura 7.9 mostra o funcionamento deste esquema. Ele está dividido em 5 fases, descritas a seguir.

Na Fase 1, é feito o *parsing* de palavras, são gerados os símbolos homofônicos, e são gerados os códigos canônicos. Nesta fase, também é gerado o *Arquivo Modelo* referente ao *Arquivo Original*.

Na Fase 2, é feito o *parsing* de caracteres do *Arquivo Modelo* gerado na Fase 1. Os demais produtos desta fase são equivalentes aos da Fase 1.

Na Fase 3, através de um procedimento pré-definido, é realizada a composição do *Arquivo Modelo*, gerado na Fase 2, com a chave do usuário, dando origem a uma *Nova Chave*.

Na Fase 4, é retomado o procedimento interrompido na Fase 1, e utilizando a *Nova Chave* é realizada a cifragem do *Arquivo Original*.

Na Fase 5, é feita a cifragem do *Arquivo Modelo*, utilizando também a *Nova Chave*.

Após o procedimento descrito, para o usuário são enviados tanto o *Arquivo Codificado* quanto o *Modelo Cifrado* através de um canal inseguro. A *Nova Chave* é enviada através do canal seguro.

O procedimento de composição deve estar bem definido para que, na decodificação, seja possível a recuperação do *Arquivo Modelo*.

Uma vantagem adicional deste esquema é que a cifragem é realizada com uma chave complexa, já que ela é a composição do modelo de caracteres com a chave do usuário. As desvantagem são o aumento do tempo de codificação; e o custo da transmissão em canal seguro da *Nova Chave.*

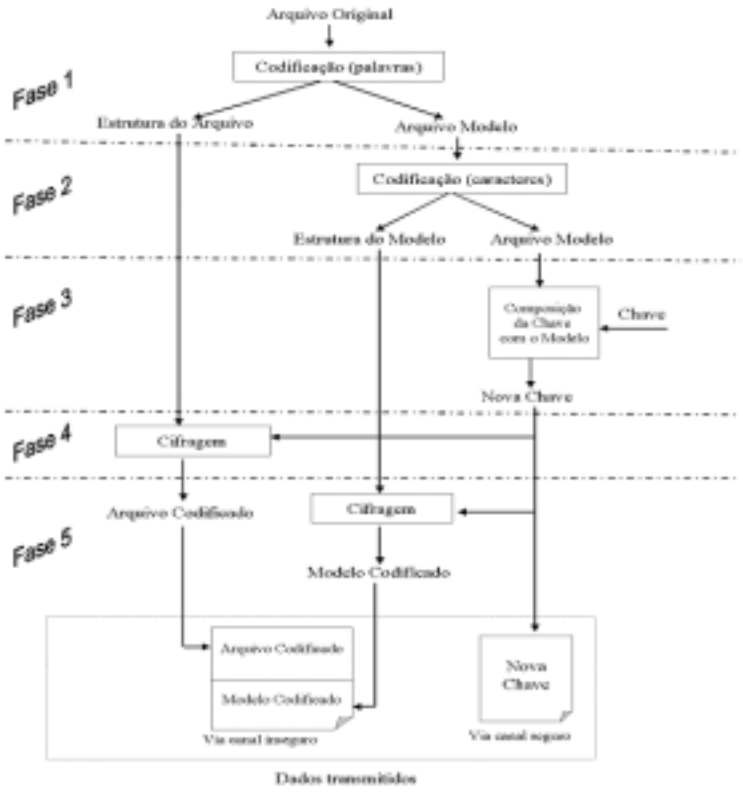

Figura 7.9: Esquema de Codificação *Chave Composta com o Modelo*.

7.6 Algoritmo Huffman Canônico

O procedimento de codificação do algoritmo *HHC* foi implementado utilizando os módulos *Parser*, *Canônico* e *Codificador*, com pequenas alterações. Para determinar os tamanhos dos códigos é utilizado o algoritmo eficiente proposto por Moffat et al. (22). Este algoritmo é uma implementação *inloco* para o algoritmo de Huffman. Com o objetivo de torná-lo ainda mais rápido, Pessoa (25) implementou algumas melhorias no algoritmo.

Para calcular os tamanhos de cada símbolo, através do algoritmo *inloco*, é necessário que as freqüências estejam ordenadas. Então, é utilizado o *Quikcsort* para ordená-las. Este algoritmo de ordenação tem custo médio de *O(n log n)*, isto significa que a ordenação não estará comprometendo a eficiência do processamento.

Na decodificação, foram utilizados os módulos *Gera Tabela de Decodificação* e *Decodificador* do algoritmo *HHC*, com algumas alterações.

7.7 Resultados Experimentais

7.7.1 Introdução

Neste capítulo, são apresentados os resultados dos testes realizados com o algoritmo *HHC*, comparando-o com o *Huffman Canônico*. É apresentada também a justificativa para o truncamento em 8 (oito) símbolos homofônicos para cada símbolo original. Além disso, são listadas as coleções de textos utilizadas nos testes. Os testes foram realizados considerando os esquemas *Tradicional* e *Modelo Codificado (Vulnerável ou Protegido)*.

7.7.2 Coleções e Ambiente

Nesta seção, são listados os arquivos de texto utilizados nos experimentos, bem como o ambiente computacional no qual os testes foram realizados.

Na tabela 7.1, encontra-se a descrição dos *arquivos de teste* utilizados nos experimentos. As coleções de José de Alencar e de Aluísio de Azevedo estão disponíveis na Biblioteca Virtual da Escola do Futuro <http://www.futuro.usp.br>. O Projeto Gutenberg é uma coleção de textos resultante de um projeto de digitação de textos. Os arquivos deste projeto estão disponíveis no endereço <ftp://ftp.sudval.org/gutenberg>, no formato texto.

Tabela 7.1: Descrição dos *arquivos de teste*.

Nome do arquivo (idioma)	**Conteúdo**	**Local de origem**
alencar.txt (português)	Coleção das obras de José de Alencar	Biblioteca Virtual do Estudante Brasileiro <www.bibvirt.futuro.usp.br>
aluisio.txt (português)	Coleção das obras de Aluísio de Azevedo	Biblioteca Virtual do Estudante Brasileiro <www.bibvirt.futuro.usp.br>
0drvb10.txt (inglês)	The Holy Bible, Douay-Rheims Version	Projeto Gutenberg <promo.net/pg>
csnva.txt (inglês)	The Complete Memoirs of Casanova	Projeto Gutenberg <promo.net/pg>
kjv10.txt (inglês)	The King James Bible	Projeto Gutenberg <promo.net/pg>
taofj10.txt (inglês)	The Antiquities of the Jews	Projeto Gutenberg <promo.net/pg>

O arquivo *alencar.txt* é composto pelas seguintes obras de José de Alencar:

- *Iracema*;
- *Cinco Minutos*;
- *Diva*;
- *Encarnação*;
- *A pata da gazela*;
- *O Guarani*;
- *Lucíola*;
- *Senhora*;
- *Til*;
- *Ubirajara*;
- *A Viuvinha*.

As seguintes obras de Aluísio de Azevedo compõem o arquivo *aluisio.txt*:

– *O Cortiço*;

– *Livro de uma Sogra*;

– *A Mortalha de Alzira*;

– *O Mulato*;

– *Casa de Pensão.*

O algoritmo está implementado em C, utilizando a ferramenta Visual C++ 6.0. O equipamento utilizado nos experimentos possui as seguintes características:

Processador:	AMD K6-II 400MHz
Sistema Operacional:	Microsoft Windows 98
Memória RAM:	256 MB
Fabricante:	Scopus

7.7.3 Definição do Número de Símbolos Homofônicos

Foi utilizada a decomposição das probabilidades de cada símbolo em potências negativas de 2, para determinar os símbolos homofônicos. Dependendo do valor das probabilidades alguns problemas podem surgir. Quando a probabilidade não é dízima periódica, a decomposição pára, mas é possível ter grande quantidade de símbolos homofônicos. Quando é dízima periódica, a decomposição não pára.

A solução simples para resolver estes problemas é parar a decomposição quando um número máximo de símbolos homofônicos for atingido. Entretanto, deve-se levar em consideração que é necessário manter as propriedades da Substituição Homofônica de Tamanho Variável.

Utilizando os *arquivos de teste*, fi feita uma simulação variando o número máximo de símbolos homofônicos na decomposição de cada símbolo original. O número de símbolos homofônicos variou de 1 até 20. Nesta simulação, foi utilizada uma chave aleatória de 512 bits e o *parsing* de palavras.

Na tabela 7.4, encontram-se os valores obtidos para o arquivo *alencar.txt*. As tabulações correspondentes aos demais arquivos encontram-se no Apêndice A. Com o objetivo de obter uma melhor precisão nos resultados, os números listados são uma média de 10 (dez) execuções idênticas do algoritmo.

A velocidade de decodificação foi determinada pela razão entre a coluna *Total* e o *Tempo de Codificação*. A *Perda de Probabilidade* representa a soma de todas as probabilidades que deixaram de ser decompostas.

Nas figuras 7.10, 7.11 e 7.12, encontram-se representados graficamente os dados referentes a simulação do arquivo *alencar.txt*. Para os demais arquivos, as tabelas e os gráficos apresentam características bastante similares.

Figura 7.10: Velocidade de Codificação do arquivo *alencar.txt*.

Figura 7.11: Aumento do tamanho do tamanho do *Arquivo Codificado* e *Arquivo Modelo* para o arquivo *alencar.txt*.

Observando-se os três gráficos nota-se que a partir de 6 (seis) símbolos homofônicos o tamanho do arquivo quase se estabiliza e a perda de probabilidades torna-se desprezível.

O único gráfico que continua com variações expressivas é o que mede a velocidade de codificação, mostrando que o aumento do número de homofônicos degrada consideravelmente o tempo de codificação, sem acrescentar nenhum beneficio ao esquema.

Então, levando em conta as observações acima e considerando uma margem de segurança, foi considerado para os experimentos um limite empírico de 8 (oito) símbolos homofônicos para cada símbolo. Isto significa que a decomposição, para cada símbolo original, é truncada em 8 símbolos homofônicos, ou antes, se toda a sua probabilidade já tiver sido decomposta em potências negativas de 2.

Figura 7.12: Perda de Probabilidade do arquivo *alencar.txt*.

7.7.4 Experimentos Realizados

Neste seção, são apresentados os experimentos realizados com o algoritmo *HHC*, adotando 8 como o número máximo de homofônicos para cada símbolo original. Em todos os experimentos foi utilizada uma chave aleatória de 512 bits.

Para mostrar que o algoritmo *HHC* não compromete a compressão dos dados, nem o tempo de codificação e de decodificação, o *HHC* foi comparado com o algoritmo *Huffman Canônico*, que é um compressor comum.

Nos experimentos, a *Taxa de Compressão* é a razão entre o tamanho do arquivo que será transmitido e o tamanho do arquivo original. Assim, a taxa de compressão mede os tamanhos do *Arquivo Codificado* e do *Arquivo Modelo* em relação ao tamanho do *Arquivo Original*.

Para estimar a perda de compressão devido ao *HHC*, foi calculada a diferença entre as duas taxas de compressão. Uma outra alternativa, é calcular a razão entre as duas taxas de compressão. Nas avaliações feitas, foi adotada a primeira estimativa.

Esquema Tradicional

A primeira comparação é realizada utilizando o esquema de codificação *Tradicional*, isto é, *parsing* de palavras no *Arquivo Original* e transmissão em aberto do *Arquivo Modelo*. As tabelas 7.6 e 7.8 ilustram essa comparação.

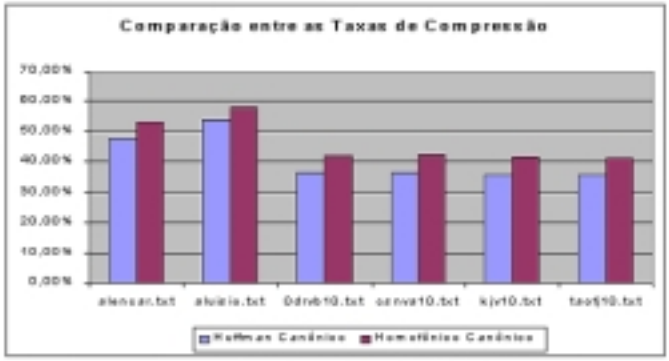

Figura 7.13: Comparação entre as taxas de compressão do *HHC* e do *Huffman*

Por exemplo, o arquivo *aluisio.txt* tem a pior nos dois algoritmos e o arquivo *taofj10.txt* a melhor, também nos dois algoritmos. Os outros arquivos também mantém as mesmas posições simultaneamente nos dois algoritmos.

Na tabela 7.10, encontram-se os tempos de codificação e decodificação dos dois algoritmos. Na codificação, o algoritmo *HHC* mostra-se mais rápido nos arquivos de língua portuguesa, possivelmente devido ao léxico ser mais extenso. Nos demais arquivos, considerando as pequenas diferenças entre os tempos, pode-se considerar que houve empate.

Na decodificação, o algoritmo *Huffman Canônico*, como esperado, tem melhor performance em todos os arquivos. Porém, a diferença é muito pequena, em torno de 1 (um) segundo. Assim, pode-se considerar que o processo de decodificação do algoritmo *HHC* também é extremamente rápido.

Na tabela 7.12, está listado o tempo gasto por cada módulo do algoritmo *HHC* na codificação. Os maiores consumidores de tempo são o *Parser* e o *Codificado*r.

Esquema Modelo Codificado

Neste esquema, o procedimento é feito em duas etapas (*Fase 1* e *Fase 2*). A primeira é idêntica ao procedimento da seção anterior. Na segunda fase, o *Arquivo Modelo* é codificado utilizando-se o *parsing* de caracteres. Assim, nesta seção, são utilizados os dados da seção anterior, complementando-os com os resultados da segunda fase deste esquema.

Na comparação, foi mantida a utilização do algoritmo *Huffman Canônico*. Cada um deles codifica seu respectivo *Arquivo Modelo*.

As tabelas 7.14 e 7.16 apresentam os dados referentes à codificação dos modelos. Considerando um mesmo arquivo original, nota-se que o *Arquivo Modelo* gerado por cada algoritmo tem tamanhos similares. Por exemplo, o arquivo *alencar.txt*, no *Huffman Canônico* o modelo tem 500.025 bytes, no *HHC* tem 463.452 bytes. Porém, neste caso, o *Huffman Canônico* tem taxas de compressão mais atraentes.

Na tabela 7.18, estão listados os tempos de codificação e decodificação desta fase. Nota-se que o processamento é bastante rápido e que os tempos dos dois algoritmos estão muito próximos. O tempo de decodificação é praticamente o mesmo para os dois algoritmos.

Nas tabelas 7.20, 7.22 e 7.24, são consolidadas as informações dos testes das duas fases deste esquema. Os tempos de codificação e decodificação não comprometem significativamente a eficiência do processo. As novas taxas de compressão também não se alteram significantemente.

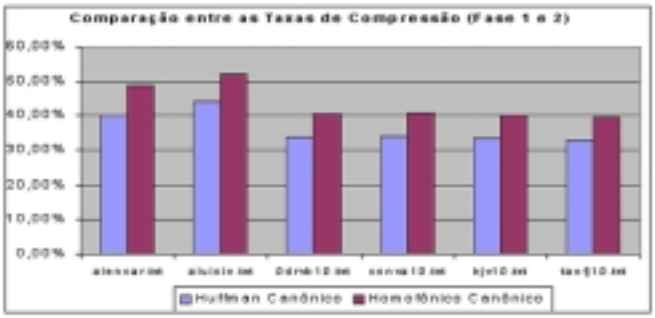

Figura 7.14: Comparação entre as taxas de compressão do *Homofônico Canônico* e do *Huffman Canônico* (Fase 1 e 2).

A grande vantagem deste processamento adicional é ter reduzido o modelo para um tamanho que permite sua transmissão através de um canal seguro a um custo relativamente pequeno, devido ao seu reduzido tamanho.

Tabela 7.4: Simulação do número de homofônicos com o arquivo *alencar.txt.*.

Número de Homofônicos	**Tempo de Codificação (segundos)**	**[1] Arquivo Codificado (bytes)**	**[2] Arquivo Modelo (bytes)**	**Total = [1] + [2] (bytes)**	**Velocidade de Codificação (KB/seg)**	**Perda de Probabilidade**
1	13,09	949.237	463.237	1.412.474	105,35	0,388799667358
2	13,52	993.873	463.283	1.457.156	105,29	0,156779333949
3	14,11	1.022.224	463.299	1.485.523	102,80	0,055923260748
4	15,02	1.038.408	463.333	1.501.741	97,66	0,012315253727
5	16,47	1.044.068	463.365	1.507.433	89,40	0,004232159117
6	16,42	1.046.844	463.385	1.510.229	89,85	0,001083814132
7	16,41	1.047.692	463.416	1.511.108	89,95	0,000204242097
8	16,64	1.047.739	463.452	1.511.332	88,68	0,000063168744
9	17,03	1.047.888	463.487	1.511.375	86,66	0,000017470051
10	17,58	1.047.940	463.514	1.511.454	83,96	0,000006075599
11	18,02	1.047.924	463.566	1.511.490	81,91	0,000002066411
12	19,14	1.047.944	463.593	1.511.537	77,14	0,000000782698
13	19,22	1.047.950	463.627	1.511.577	76,82	0,000000315005
14	19,88	1.047.783	463.668	1.511.451	74,24	0,000000123027
15	20,32	1.047.924	463.700	1.511.624	72,64	0,000000042565
16	21,61	1.048.044	463.722	1.511.766	68,31	0,000000006364
17	22,05	1.048.034	463.740	1.511.774	66,96	0,000000002603
18	23,77	1.047.883	463.758	1.511.641	62,11	0,000000001066
19	23,31	1.047.879	463.766	1.511.645	63,33	0,000000000410
20	24,10	1.047.941	463.784	1.511.725	61,27	0,000000000119

Tabela 7.6: Codificação do *Arquivo Original* com o *Huffman Canônico*.

Huffman Canônico (Codificação do arquivo)					
Arquivo	**Arquivo Original (bytes)**	**[1] Arquivo Codificado (bytes)**	**[2] Arquivo Modelo (bytes)**	**Total = [1]+[2] (bytes)**	**Taxa de Compressão (%)**
alencar.txt	2.843.945	849.869	500.025	1.349.894	47,47%
aluisio.txt	1.911.875	581.321	446.612	1.027.933	53,77%
0drvb10.txt	5.910.467	1.785.768	355.472	2.141.240	36,23%
csnva10.txt	6.872.548	2.079.684	416.919	2.496.603	36,33%
kjv10.txt	4.445.260	1.328.490	258.048	1.586.538	35,69%
taofj10.txt	3.008.331	863.029	206.003	1.069.032	35,54%

Tabela 7.8: Codificação do *Arquivo Original* com o *HHC*.

HHC (Codificação do arquivo)					
Arquivo	**Arquivo Original (bytes)**	**(1) Arquivo Codificado (bytes)**	**(2) Arquivo Modelo (bytes)**	**Total = (1)+(2) (bytes)**	**Taxa de Compressão (%)**
alencar.txt	2.843.945	1.047.739	463.452	1.511.191	53,14%
aluisio.txt	1.911.875	692.960	413.529	1.106.489	57,87%
0drvb10.txt	5.910.467	2.147.992	329.918	2.477.910	41,92%
csnva10.txt	6.872.548	2.510.344	389.048	2.899.392	42,19%
kjv10.txt	4.445.260	1.606.301	237.398	1.843.699	41,48%
taofj10.txt	3.008.331	1.044.647	192.026	1.236.673	41,11%

Tabela 7.10: Tempos de Codificação e Decodificação.

	Codificação		**Decodificação**	
Arquivo	***Huffman Canônico*** *(segundos)*	***HHC*** *(segundos)*	***Huffman Canônico*** *(segundos)*	***HHC*** *(segundos)*
alencar.txt	32,17	16,64	4,45	5,11
aluisio.txt	24,03	12,79	3,07	3,49
0drvb10.txt	29,56	29,71	8,99	9,91
csnva10.txt	36,07	34,24	10,41	11,52
kjv10.txt	20,21	22,64	6,72	7,32
taofj10.txt	13,46	14,09	4,41	4,88

Tabela 7.12: Tempos de Codificação por módulo.

	Algoritmo HHC				
Arquivo	**Módulo Parser (segundos)**	**Módulo Homofônico (segundos)**	**Módulo Canônico (segundos)**	**Módulo Gera Permutações (segundos)**	**Módulo Codificador (segundos)**
alencar.txt	4,61	2,29	1,24	0,52	7,98
aluisio.txt	3,50	2,09	1,06	0,46	5,68
0drvb10.txt	9,07	1,98	1,17	0,39	17,09
csnva10.txt	10,34	2,07	1,21	0,44	20,17
kjv10.txt	7,05	1,43	0,76	0,33	13,06
taofj10.txt	4,43	1,02	0,63	0,21	7,79

Tabela 7.14: Codificação do *Arquivo Modelo* usando *Huffman Canônico*.

Huffman Canônico (Codificação do modelo)					
Arquivo	**Arquivo Modelo (bytes)**	**[1] Modelo Codificado (bytes)**	**[2] Novo Arquivo Modelo (bytes)**	**Total = [1]+[2] (bytes)**	**Taxa de Compressão (%)**
alencar.txt	500.025	290.720	641	291.361	58,27%
aluisio.txt	446.612	260.873	687	261.560	58,57%
0drvb10.txt	355.472	208.774	519	209.293	58,88%
csnva10.txt	416.919	245.760	522	246.282	59,07%
kjv10.txt	258.048	149.916	508	150.424	58,29%
taofj10.txt	206.003	122.673	520	123.193	59,80%

Tabela 7.16: Codificação do *Arquivo Modelo* usando o *HHC*.

HHC (Codificação do modelo)					
Arquivo	**Arquivo Modelo (bytes)**	**[1] Modelo Codificado (bytes)**	**[2] Novo Arquivo Modelo (bytes)**	**Total = [1]+[2] (bytes)**	**Taxa de Compressão (%)**
alencar.txt	463.452	335.423	934	336.357	72,58%
aluisio.txt	413.529	299.371	948	300.319	72,62%
0drvb10.txt	329.918	239.918	799	240.717	72,96%
csnva10.txt	389.048	285.490	828	286.318	73,59%
kjv10.txt	237.398	172.276	820	173.096	72,91%
taofj10.txt	192.026	140.787	781	141.568	73,72%

Tabela 7.18: Tempo de Codificação do *Arquivo Modelo*.

	Codificação		**Decodificação**	
Arquivo	***Huffman Canônico*** *(segundos)*	***HHC*** *(segundos)*	***Huffman Canônico*** *(segundos)*	***HHC*** *(segundos)*
alencar.txt	2,82	2,92	1,37	1,38
aluisio.txt	2,36	2,70	1,23	1,23
0drvb10.txt	2,03	2,20	0,99	0,99
csnva10.txt	2,37	2,70	1,16	1,16
kjv10.txt	1,43	1,59	0,71	0,71
taofj10.txt	1,15	1,45	0,55	0,56

Tabela 7.20: Final da Codificação usando *Huffman Canônico*.

Huffman Canônico (Codificação Arquivo Original e Arquivo Modelo)						
Arquivo	**Tamanho Original (bytes)**	**[1] Arquivo Codificado (bytes)**	**[2] Modelo Codificado (bytes)**	**[3] Novo Arquivo Modelo (bytes)**	**Total = [1]+[2]+[3] (bytes)**	**Taxa de Compressão Final (%)**
alencar.txt	2.843.945	849.869	290.720	641	1.141.230	40,13%
aluisio.txt	1.911.875	581.321	260.873	687	842.881	44,09%
0drvb10.txt	5.910.467	1.785.768	208.774	519	1.995.061	33,75%
csnva10.txt	6.872.548	2.079.684	245.760	522	2.325.966	33,84%
kjv10.txt	4.445.260	1.328.490	149.916	508	1.478.914	33,27%
taofj10.txt	3.008.331	863.029	122.673	520	986.222	32,78%

Tabela 7.22: Final da Codificação usando o *HHC*.

HHC (Codificação Arquivo Original e Arquivo Modelo)						
Arquivo	**Tamanho Original (bytes)**	**[1] Arquivo Codificado (bytes)**	**[2] Modelo Codificado (bytes)**	**[3] Novo Arquivo Modelo (bytes)**	**Total = [1]+[2]+[3] (bytes)**	**Taxa de Compressão Final (%)**
alencar.txt	2.843.945	1.047.739	335.423	934	1.384.096	48,67%
aluisio.txt	1.911.875	692.960	299.371	948	993.279	51,95%
0drvb10.txt	5.910.467	2.147.992	239.918	799	2.388.709	40,41%
csnva10.txt	6.872.548	2.510.344	285.490	828	2.796.662	40,69%
kjv10.txt	4.445.260	1.606.301	172.276	820	1.779.397	40,03%
taofj10.txt	3.008.331	1.044.647	140.787	781	1.186.215	39,43%

Tabela 7.24: Tempos de Codificação e Decodificação Finais.

	Codificação		**Decodificação**	
Arquivo	***Huffman Canônico*** *(segundos)*	***HHC*** *(segundos)*	***Huffman Canônico*** *(segundos)*	***HHC*** *(segundos)*
alencar.txt	34,99	19,56	5,82	6,49
aluisio.txt	26,39	15,49	4,30	4,72
0drvb10.txt	31,59	31,91	9,98	10,90
csnva10.txt	38,44	36,94	11,57	12,68
kjv10.txt	21,64	24,23	7,43	8,03
taofj10.txt	14,61	15,54	4,96	5,44