Implementação do Código de Hamming
Algoritmo de Correção de Erros

Pedro Henrique Marques de Lima , Felipe Gonçalves Pereira

[1] Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

{phml23,fgp23}@inf.ufpr.br

Resumo. *Este trabalho apresenta a implementação do código de Hamming (31,26) para codificação e decodificação de arquivos, permitindo a detecção e correção automática de erros únicos em blocos de dados. A implementação em linguagem C é eficiente e adequada para proteção contra erros de transmissão ou armazenamento, oferecendo um equilíbrio entre redundância e capacidade de correção.*

1. Introdução

O código de Hamming é um algoritmo de correção de erros criado por Richard Hamming em 1950. Este trabalho descreve a implementação em C do código de Hamming (31,26), que codifica blocos de 26 bits adicionando 5 bits de paridade para formar palavras-código de 31 bits, capazes de detectar e corrigir erros simples em arquivos digitais.

2. Fundamentos Teóricos

2.1. Código de Hamming (31,26)

O código utiliza 26 bits de dados e 5 bits de paridade, posicionados nas potências de 2 (1, 2, 4, 8 e 16). Os bits de paridade são calculados para garantir a detecção e correção de erros em um único bit.

2.2. Detecção e Correção

Durante a codificação, os bits de paridade são calculados via operações XOR. Na decodificação, a posição do erro é localizada usando XOR dos bits recebidos, permitindo a correção automática de erros únicos.

3. Implementação

3.1. Estrutura Principal

O programa oferece duas operações principais: codificação, que converte o arquivo original em uma versão codificada; e decodificação, que recupera o arquivo original corrigindo possíveis erros.

3.2. Função de Codificação

A função `codifica()` insere bits de paridade nas posições corretas:

```c
uint32_t codifica (uint32_t block) {
    unsigned int codded = 0;
    unsigned int xor = 0;

    for (int i = 0, pot = 1, j = 0; i < 31; i++) {
        if ((i+1) == pot) {
            pot *= 2;
        } else {
            unsigned int bit = (block >> j) & 1;
            codded |= (bit << i);
            if (bit) xor ^= (i+1);
            j++;
        }
    }

    for (int pot = 1, i = 0; i < 5; pot *= 2, i++) {
        codded |= (((xor >> i) & 1) << (pot - 1));
    }
    return codded;
}
```

3.3. Função de Decodificação

A função `decodifica()` identifica e corrige erros únicos:

```c
int32_t decodifica (unsigned long int block) {
    unsigned long int xor = 0;

    for (int i = 0; i < 31; i++) {
        if ((block >> i) & 1)
            xor ^= (i+1);
    }

    if (xor)
        block ^= 1 << (xor-1);
}
```

4. Processamento de Arquivos

4.1. Codificação

O arquivo é lido em buffers de vários bytes, que são divididos em chunks de 26 bits para codificação via código de Hamming, armazenando-se o tamanho original codificado em 26 bits para reconstrução.

4.2. Decodificação

Na decodificação, o arquivo codificado é lido linha a linha, aplicando-se a correção e reconstrução do arquivo original respeitando o tamanho armazenado.

5. Características da Implementação

5.1. Manipulação de Bits

A implementação faz uso eficiente de operações bit a bit e buffers para manipulação e armazenamento dos dados codificados.

5.2. Formato de Arquivo Codificado

O arquivo codificado inclui o tamanho original na primeira linha e blocos codificados nas linhas subsequentes em formato texto, separados por quebras de linha.

6. Vantagens e Limitações

6.1. Vantagens

- Correção automática de erros únicos
- Detecção de erros duplos
- Eficiência computacional em C
- Processamento por blocos para arquivos grandes

6.2. Limitações

- Overhead de 23% devido aos bits extras
- Correção limitada a erros únicos
- Erros múltiplos podem não ser detectados
- Limitação do tamanho máximo do arquivo (64 MB)

7. Conclusão

A implementação do código de Hamming (31,26) mostrou-se eficiente para correção de erros simples em arquivos digitais, combinando baixo overhead com capacidade adequada para aplicações que demandam proteção contra erros isolados em sistemas digitais.