Detector e Corretor de Erros

Calvin Ferreira da Rosa, Elias Gabriel Grasel, Felipe Augusto Hertzer

Disciplina de Comunicação de Dados
Universidade de Santa Cruz do Sul (UNISC) – Santa Cruz do Sul, RS – Brasil

eliasgrasel@mx2.unisc.br, calvin@mx2.unisc.br felipeaugusto@mx2.unisc.br

abstract. *This article describes the implementation of hamming code of exemplifying some of your operation and detection mode and error correction. The same is used in telecommunications in the signal processing and storage of secure and efficient ways.*

resumo. *Este artigo descreve a implementação do código de hamming exemplificando um pouco do seu funcionamento e modo de detecção e correção de erros. O mesmo é usado nas telecomunicações no processamento de sinal e no armazenamento de formas seguras e eficiente.*

1.Introdução

Em informática, telecomunicações, acontecem erros nas mensagens transmitidas, fazendo com que esta seja modificada para um valor que não corresponde ao do original, o que é extremamente inconveniente. A detecção e correção de erros em uma comunicação de dados é muito importante para que esta seja feita de uma forma correta e sem falhas. O código de Hamming é um código de bloco linear, desenvolvido por Richard Hamming, e utilizado no processamento de sinal e nas telecomunicações. A sua utilização permite a transferência e armazenamento de dados de forma segura e eficiente. Nas telecomunicações os códigos de Hamming utilizados são generalizações do Hamming.

2.Funcionamento o código de Hamming

Os códigos de Hamming tem a função de detectar erros de até dois bits e corrigir até um bit. O código de paridade não pode corrigir erros, e só pode detectar apenas um número ímpar de erros. Devido à sua simplicidade os códigos Hamming, são amplamente utilizados na memória dos computadores. Neste contexto, é frequente utiliza um código de Hamming estendido com um bit de paridade extra.

Em termos matemáticos, códigos de Hamming são uma classe de códigos lineares binários. Para cada inteiro existe um código de comprimento de bloco comprimento de bloco e com comprimento de mensagem. Por isso, a taxa de códigos de Hamming é, o que é a mais alta possível para códigos com distância e de comprimento de bloco. A matriz de paridade de um código de Hamming é construída listando todas

as colunas de comprimento que são linearmente independentes. Os códigos de Hamming são considerados códigos perfeitos, isto é, alcançam a taxa mais alta para os códigos com o seu comprimento de bloco e uma distância mínima.

3. Desenvolvimento

3.1 Conversão ASCII para binário e de binário para ASCII

Neste trecho do código é feita a conversão de cada caractere para binários, baseado em sua sequência binária conforme tabela ASCII.

```php
// Implementação da conversão ASCII - BINÁRIO - valor 1.
function str2bin($input)
{
    if (!is_string($input)) return null;
    $value = unpack('H*', $input);
    return str_pad(base_convert($value[1], 16, 2), 8, "0", STR_PAD_LEFT);
}
```

```php
// Implementação da conversão BINÁRIO - ASCII - valor 1.
function bin2str($input)
{
    if (!is_string($input)) return null;
    return pack('H*', base_convert($input, 2, 16));
}
```

3.2 Cálculo da paridade

O trecho a seguir demonstra como é feito o cálculo da paridade, onde neste caso é passado como parâmetro o valor já em binário através de um array, acompanhado de o tipo de paridade escolhida.

Neste caso é informado o tipo de paridade par ou ímpar, e se é do tipo vertical ou horizontal. O bit é o bit da paridade, que por padrão vem como 0 e caso após percorrer todo o array, o valor restante for 1, o bit receberá 1 para que possa completar a quantidade de 1 necessária para a paridade par, senão, continuará sendo atribuído o 0.

Bit de Paridade - Par - VRC + HRC

1	2	3	4	5	6	7	8	Paridade	Letra
0	1	0	1	0	1	0	1	0	U
0	1	0	0	1	1	1	0	0	N
0	1	0	0	1	0	0	1	1	I
0	1	0	1	0	0	1	1	0	S
0	1	0	0	0	0	1	1	1	C
0	1	0	0	0	0	1	0		

3.3 CRC

No trecho de código a seguir, será demonstrado a verificação CRC. Na primeira função é onde o loop vai trabalhar com cada posição. Já dentro do loop transformamos a mensagem inicial em binário e logo após retiramos os zeros a esquerda e adicionamos zeros no fim da mensagem, se o CRC for de 8, vamos adicionar 8 zeros no final.

Logo após vamos dividir a mensagem pelo polinômio escolhido. Dentro da função *calcula_crc,* verifica na posição da mensagem e na posição do polinômio atual do loop, se o valor de ambos forem igual a mensagem recebe zero naquela posição e naquele momento, se forem diferentes recebe 1. Assim até atingir o mesmo tamanho do polinômio. Se o tamanho for o mesmo isso quer dizer que temos que passar para a próxima linha, e então fazemos uma função recursiva para calcular com a nova mensagem gerada e o polinômio, isso até que o resto da mensagem for menor ou igual ao tamanho do polinômio, se for menor então já temos a resposta.

```php
// Itens 1 a 7 mais implementar modelo de detecção de erros com CRC - valor 2 pontos extra
function crc($array_string, $polinomio){
    $polinomios = self::polinomio($polinomio);
    $array_retorno = array();
    foreach($array_string as $key => $a) {
        $binario = self::str2bin($a);
        $mensagem = ltrim($binario.str_pad("", strlen($polinomios) - 1, "0", STR_PAD_LEFT), "0");
        $array_retorno[$key] = str_split($binario.self::calcula_crc($mensagem, $polinomios));
    }
    return $array_retorno;
}

function calcula_crc($mensagem, $polinomio){
    $mensagem_array = str_split($mensagem);
    $polinomio_array = str_split($polinomio);
    $count = 0;
    $resultado = "";
    foreach($mensagem_array as $m){
        if(strlen($polinomio) == $count){
            $resto = ltrim($resultado.substr($mensagem, $count), "0");
            $resultado = ltrim($resultado.substr($mensagem, $count), "0");
            if(strlen($resto) >= strlen($polinomio))
                $resultado = self::calcula_crc($resto, $polinomio);
            break;
        } else {
            $resultado .= ($m == $polinomio_array[$count] ? "0" : "1");
            $count++;
        }
    }
    return str_pad(ltrim($resultado, "0") , strlen($polinomio) -1, "0", STR_PAD_LEFT);
}
```

CRC - 8 Bits

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Letra
0	1	0	1	0	1	0	1	1	0	1	0	1	1	0	0	U
0	1	0	0	1	1	1	0	1	1	1	0	1	1	0	1	N
0	1	0	0	1	0	0	1	1	1	1	1	1	0	0	0	I
0	1	0	1	0	0	1	1	1	0	1	1	1	1	1	0	S
0	1	0	0	0	0	1	1	1	1	0	0	1	1	1	0	C

Nesta parte, somente são atribuídos os valores dos polinômios geradores, conforme os bits escolhidos.

```php
function polinomio($p){
    $polinomios = array(
        "5" => "110101",
        "8" => "100000111",
        "12" => "1000000001011",
        "16" => "11000000000000101",
        "32" => "100000100110000010001110110110111",
    );
    return (isset($polinomios[$p]) ? $polinomios[$p] : $polinomios[8]);
}
```

3.5 Hamming

No hamming desenvolvemos a primeira função onde adiciona zeros a mensagem original nas posições ondem ficam os multiplicadores com o 1,2,4,8 e etc. E logo depois chama a função *hamming_verifica()*.

```php
// Implementação da codificação de Hamming - valor 2.
function hamming($array_string){
    $array_retorno= array();
    foreach($array_string as $k => $a){
        $binario = self::str2bin($a);
        $array_binario = str_split($binario);
        // Adiciona os bits de paridade com zero
        foreach(self::asciitoArrayBin($array_binario) as $key => $value) {
            $posicao = $key + 1;
            if(($posicao & ($posicao - 1)) == 0){
                self::array_insert($array_binario, $key, 0);
            }
        };
        // Faz o encoding
        $array_retorno[$k] = self::hamming_verifica($array_binario);
    }
    return $array_retorno;
}
```

Na próxima imagem podemos ver a primeira função onde corrige a mensagem recebida e gera uma nova mensagem com os dados recebidos e onde compara os dois resultados.

Nessa comparação se for diferente algum bit, ele soma a posição de todos os 1's dos multiplicadores da linha que houve o problema. E assim saberá o bit que está incorreto.

```php
public function verificar_erro_hamming($matriz)
{
    $array_retorno= $array_retorno_original= array();
    foreach($matriz as $key => $a){
        $retorno = array_map('intval', self::hamming_verifica($a));
        $array_retorno[$key] = $retorno;
        $array_retorno_original[$key] = array_map('intval',$a);
    }
    return array("verificado" => $array_retorno, "original" => $array_retorno_original);
}

function hamming_verifica($array_binario){
    $array_editado = $array_binario;
    foreach($array_binario as $k => $b){
        $x = $k + 1;
        if(($x & ($x - 1)) == 0){
            $resultado = 0;
            $last = $count = 0;
            $proximo = $x - 1;
            foreach($array_binario as $t => $m){
                $resultado = ($last != $x ? 0 : $resultado);
                if($x == 1 && ($t+1) & 1){
                    if($t > 0){
                        $resultado = $resultado + ($m == 1 ? 1 : 0);
                    }
                } else if($x != 1 && $t >= ($x - 1)){
                    if($count < $x && $t >= $proximo){
                        if($x != ($t + 1)){
                            $resultado = $resultado + ($m == 1 ? 1 : 0);
                        }
                        $count++;
                    } else if($count > 0){
                        $count = 0;
                        $proximo = $t + $x;
                    }
                }
                $last = $x;
            }
            $array_editado[$k] = $resultado & 1 ? 1 : 0;
        }
    }
    $array_editado = array_map('intval', $array_editado);
    return $array_editado;
}
```

A função *hamming_verifica* recebe o vetor já com os zeros nas posições dos multiplicadores, e no primeiro loop é verificado se a posição atual é uma multiplicador (1,2,4,8...), no outro loop, temos um *if* que verifica se a posição for igual a 1 entrando nela a variável $resultado recebe mais 1 se o valor atual for 1 e 0 se o valor atual for zero. Nas outras execuções do loop verificamos se passou os números de posições que deve pular, por exemplo, se o multiplicador for 2 deverá pular duas posições para entrar no *if* novamente e somar a variável $resultado, no final verificamos se a soma da variável resultado for ímpar, o valor da posição do multiplicador atual recebe 1 e se for par recebe 0. A variável $proximo recebe a posição atual mais a posição do multiplicador que está atuando, por exemplo para o

multiplicador 4 e posição 5, vamos ter a variável próximo recebendo 5 + 4, e então só será calculado novamente a variável $resultado quando atingir a posição 9.

Na próxima imagem demonstra a página onde fizemos alguns cálculos finais do Hamming e também a contagem de posições onde os bits da mensagem recebida forem diferentes da mensagem recalculada.

```
<table class="table table-bordered table-hover table-condensed">
    <thead>
            <tr>
                <?php
                foreach($data['retorno'][key($data['retorno'])] as $chave => $value){ ?>
                    <th class="active text-center"><?php echo $chave+ 1; ?></th>
                <?php } ?>
                <th class="active text-center">Status</th>
                <th class="active text-center"><b>Letra</b></th>
            </tr>
    </thead>
<?php foreach ($data['retorno'] as $letra => $binario) { ?>
    <tr>
        <?php $count = 0;
        foreach($binario as $chave => $value){ $p = $chave + 1;
            if(($p & ($p - 1)) == 0){
                $efeitos = "text-danger danger";
                if($data['retorno_original'][$letra][$chave] != $value){
                    $count = $count + $p;
                }
            } else {
                $efeitos = "";
            } ?>
            <td class="text-center <?php echo $efeitos; ?>"><?php echo $value; ?></td>
        <?php } ?>
        <td class="<?php echo ($count == 0 ? "success" : "danger"); ?> text-center"><?php echo ($count == 0 ? "Ok" : "Erro ".$count); ?></td>
        <td class="active text-center"><b><?php
                $letra = "";
                foreach($binario as $key => $b){
                    $posicao = $key + 1;
                    $letra .= (($posicao & ($posicao - 1)) == 0 ? "" : $b);
                }
                echo utf8_encode(\Helpers\Basicas::bin2str($letra));
                ?></b></td>
    </tr>
<?php } ?>
</table>
```

4. Conclusão

Podemos concluir que o código de Hamming tem uma grande importância nas telecomunicações, visto que o mesmo permite a detenção e localização de um bit errado, sendo que as telecomunicações usam transmissão digital de dados, isto é, caso uma dada sequência de bits tenha um bit errado o Hamming permite a detecção, localização e correção do mesmo.

O CRC e o bit de paridade também são muito importantes pois todos eles asseguram que as mensagens recebidas foram as mesmas que foram enviadas e que o meio físico não tenha alterada por algum problema durante a transmissão. A paridade é uma detecção mais simples, onde é definido um tipo de paridade, par ou ímpar, pelo transmissor e então a partir da quantidade de 1 que a mensagem possui, é acrescentado o valor 0 ou 1, dependendo da paridade e da quantidade de 1 que a mensagem possui para ser par, ou então, ímpar. A paridade pode ser considerada vertical, horizontal ou então, para maior segurança, verificação vertical e horizontal. O cálculo CRC já é uma verificação um pouco mais complexa, onde também é definido um polinomio gerador, que será feito o cálculo da divisão da mensagem que deseja transmitir com o código binário do polinômio, e após uma verificação da mensagem recebida, com o polinômio gerador, onde o resultado precisa ser sempre zero para estar correta.

Cada um desses algoritmos tem seus prós e contras, e cada um tem o local de sua utilidade, todos são muito importantes, independentemente de sua complexidade.

5. Referencias

<https://br.answers.yahoo.com/question/index?qid=20090606083802AABCeHj>

<https://pt.wikipedia.org/wiki/C%C3%B3digo_de_Hamming>