Codificadores e decodificadores para detecção e correção de erros

Bruno Selau, Maria Eduarda Casanova

Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) – Porto Alegre – RS – Brazil

1. Introdução

O trabalho tem como objetivo a implementação de codificadores e decodificadores para detecção e correção de erros. Com isso foram implementadas as seguintes técnicas: redundância de bloco, CRC e código de Hamming. Os codificadores e decodificadores apresentados neste relatório foram implementados através da linguagem de programação funcional conhecida como python, e os códigos fontes deste relatório podem ser encontrados aqui.

2. Implementação

2.1. Redundância de bloco

2.1.1 Codificador

O codificador de redundância de blocos começa a partir do recebimento de uma palavra, após isso é necessário descobrir o código ASCII de cada letra, e então colocar transformar este código em binário. Para cada letra em ASCII binário é necessário verificar a paridade deste código, caso a quantidade de 1's seja ímpar, será adicionado outro 1 no final deste binário, caso contrário será adicionado um 0. Após isso, é criado um código binário de paridade que é adicionado ao final da codificação, este novo código é é construído de forma que é contada a paridade dos bit's, ao comparar cada ASCII.

2.1.2 Decodificado

O decodificador de redundância de blocos começa a partir do recebimento de um hexadecimal, que é tratado para se tornar um binário. este binário será dividido em *n* partes de 8 bits. Após isso será verificado a paridade dos índices, certificando-se que a *n*-ésima parte é o resultado da paridade dos índices das partes anteriores. Então acontecerá a segunda verificação, que garantirá que o oitavo bit de cada parte é resultado da paridade dos outros bits.

```
ada@MacBook-Pro-de-Maria-3 bcc % python3 bcc_encoder.py redes
E4CAC9CAE7CA
ada@MacBook-Pro-de-Maria-3 bcc % python3 bcc_decoder.py E4CAC9CAE7CA
redes
ada@MacBook-Pro-de-Maria-3 bcc % python3 bcc_decoder.py E4CAC9CAE7CB
ERRO
```

2.2. CRC

2.2.1 Codificador

O codificador CRC começa a partir do recebimento de uma palavra e de um binário gerador de ordem 5.

Após isso o os seguintes passos devem ser feitos para cada letra, começando por transformar essa letra em um um código ASCII, e transformar este código em binário e adicionar quatro 0's no final deste código. O letter_slice receberá os 5 primeiros bits da letra atual, isto será usado para as operações de xor.

Se o primeiro bit de letter_slice for 1, letter_slice receberá o resultado de um XOR entre letter_slice e o binário gerador, caso contrário receberá o resultado do XOR entre letter_slice e uma lista de 0's. Em ambos os casos é necessário preencher o 0's a esquerda. Após isso, é necessário remover o primeiro bit de letter_slice, e adicionar um novo bit ao final, esse bit é o próximo bit do binário ASCII que ainda não esteve em letter_slice.

Isto repete até todos os bits serem verificados, ao final o resultado será transformado em um hexadecimal. Estes passos devem ser feitos para todas as letras, e o resultado final é a concatenação de todos os resultados hexa.

2.2.2 Decodificador

O decodificador CRC começa a partir do recebimento de um hexadecimal e um binário gerador, este hexadecimal é dividido em *n* partes de 3 símbolos.

Cada parte de 3 símbolos passa pelo processo mesmo processo, que começa ao transformar esse 3 hexadecimais em um sequência de binários, que chamamos de binary_letter. O letter slice será preenchido pelos 5 primeiros bits do binário, após isso é necessário fazer o mesmo processo para o letter_slice que é feito no codificador. A diferença é a verificação feita em letter_slice ao final, pois o resultado está correto apenas se o letter_slice é finalizado como uma lista de 0's, neste caso será o binário inicial será convertido em hexa novamente, e será removido o último dígito, caso contrário ocorreu um erro.

Esse processo acontecerá para cada parte de 3 símbolos, o resultado final é a concatenação do resultado do CRC para cada símbolo.

```
ada@MacBook-Pro-de-Maria-3 CRC % python3 crc_encoder.py redes 10101
72365964C659736
ada@MacBook-Pro-de-Maria-3 CRC % python3 crc_decoder.py 72365964C659736 10101
redes
ada@MacBook-Pro-de-Maria-3 CRC % python3 crc_decoder.py 72365A64C659737 10101
r_de_
ERRO nos caracteres: [2, 5]
```

```
ada@MacBook-Pro-de-Maria-3 CRC % python3 crc_encoder.py pucrs 10011
70875763872E73D
ada@MacBook-Pro-de-Maria-3 CRC % python3 crc_decoder.py 70875763872E73D 10011
pucrs
ada@MacBook-Pro-de-Maria-3 CRC % python3 crc_decoder.py 70875663872E73D 10011
p_crs
ERRO nos caracteres: [2]
```

2.3. Código de Hamming

2.3.1 Codificador

O codificador de Hamming começa a partir do recebimento de uma palavra, e o algoritmo a seguir será feito para cada letra.

As primeiras operações tratam de transformar uma letra em um binário invertido, após isso uma nova lista será criada, nomeada hammingword, e o algoritmo buscará os "espaços de hamming". Para cada posição é necessário verificar se ela é uma potência de dois, caso a posição seja um potência de dois é necessário preencher a posição com "_".

Após este passo, é necessário descobrir quais posições possuem o bit 1, essas posições serão guardada em onesPositions. Conhecendo estas posições, será feito um xor entre todas as posições com 1, é necessário transformar este resultado em binário e colocar 0's a esquerda. Por fim, o algoritmo coloca os bits resultantes do xor, nos espaços que possuem "_", então, essa mensagem é transformada em hexa.

2.3.2 Decodificador

O decodificador de Hamming começa a partir do recebimento de um hexa, que será dividido em *n* partes de 3 dígitos.

Para cada parte é necessário transformar esses 3 dígitos em binário, e inverter esta lista de binários. Após isto o algoritmo busca todas posições com o bit 1, que chamaremos de onesPositions. Após isso fazer um xor entre todas estes posições com bit 1. Se o resultado do xor é diferente de 0, nós temos 1 erro. O resultado do xor indica a posição com erro, ou seja, basta inverter o resultado atual. Para ter o binário final o algoritmo pega todas as posições que não são potências de dois, e então já converte para seu símbolo em caráter.

```
ada@MacBook-Pro-de-Maria-3 HAMMING % python3 ham_encoder.py redes
79962C62B62C79E
ada@MacBook-Pro-de-Maria-3 HAMMING % python3 ham_decoder.py 79962C62B62C79E
redes
ada@MacBook-Pro-de-Maria-3 HAMMING % python3 ham_decoder.py 79961C62B62C69E
rbdes
ERRO no caractere  2  -> Correção:  b
ERRO no caractere  5  -> Correção:  s
```

3. Conclusão

Os trabalho proposto na cadeira oportunizou uma experiência prática com esses codificadores e decodificadores, proporcionando um maior entendimento sobre o que acontece nos casos de erros e falhas na troca de mensagens. Com isso foi possível compreender como são detectados os erros, além do fato de podermos compreender e implementar a correção para estes casos. Ao longo do processo encontramos a mesma dificuldade no CRC e no código de Hammimg, pois quando falado em mensagem na situação de redundância de blocos, estamos falando sobre a palavra inteira, mas ao falar de mensagem nestes dois outros algoritmos, está sendo falado de cada letra.