

# Codificador/Decodificador convolucional usando o algoritmo de Viterbi

Aristides Darlan Peiter<sup>1</sup>, Maicon Ghidolin<sup>1</sup>, Wagner Frana<sup>1</sup>

<sup>1</sup>Ciência da Computação – Universidade Federal da Fronteira Sul (UFFS)

{aristosdp, maicon.ghidolin, wagnerfrana94}@gmail.com

**Resumo.** Este artigo, consiste na descrição das principais etapas da implementação de um algoritmo de codificação e decodificação convolucional usando o algoritmo de Viterbi, sendo este, o trabalho de número dois da disciplina de Inteligência Artificial ministrada pelo professor José Carlos Bins Filho, no semestre 2017.1.

**Palavras-chave:** codificação, ruído, decodificação.

## 1. Codificador/Decodificador convolucional usando o algoritmo de Viterbi

### 1.1. Descrição geral do algoritmo

O algoritmo de codificação/decodificação convolucional usando o algoritmo de Viterbi desenvolvido nesse trabalho, consiste nas etapas de codificação, geração de ruído, decodificação e por fim comparação dos resultados obtidos. Os valores de entrada do algoritmo são passados via argumento no terminal de comandos respeitando o seguinte formato :

```
java Main BitsDeEntrada Ruído.
```

Após estabelecido os bits de entrada e o ruído associado, inicia-se a etapa de codificação dos bits. Nessa fase, cada bit de entrada influencia a geração de três pares de bits de saída. Todavia, para cada bit de entrada são gerados apenas dois bits de saída, além disso, cada bit coloca o estado atual em um dos 4 estados possíveis (00, 01, 10, 11), sendo que o estado inicial do processo é o “00”. Quando a leitura dos bits de acabar, são adicionados 2 bits extras no final da mesma, para usar toda a influência dos bits da entrada, e os mesmos passaram pelo mesmo processo que os demais bits, conforme descrito acima. As trocas de estado e o par de bits emitidos pela saída acontecem conforme a tabela abaixo:

Estado Atual	Par emitido		Estado Atual	Próximo estado	
	Ent=0	Ent=1		Ent=0	Ent=1
00	00	11	00	00	10
01	11	00	01	00	10
10	10	01	10	01	11
11	01	10	11	01	11

Tabela de Transições e Emissões

Como saída desta fase obtem-se uma cadeia de bits codificados, a partir disso, inicia-se a etapa de geração de ruído. Nessa fase, conforme o valor do ruído passado como argumento na execução do programa, sendo esse pertencente ao intervalo  $[0, 1]$ , os bits da cadeia sofrem alteração de valor. Após um sorteio de forma aleatória para cada bit, se o valor obtido for menor ao ruído, um bit da cadeia igual a “1” passa a ser “0” e um com valor “0” passa a ser “1”.

Estabelecido o ruído inicia-se a etapa de decodificação usando o algoritmo de Viterbi. A partir da mensagem de saída do gerador de ruído, começando pelo estado “00” vai-se fazendo as transições, conforme a tabela acima e calculando para cada caminho o erro acumulado. Pegando de dois em dois os bits da mensagem, o algoritmo faz o cálculo de erro gerado, esse erro é calculado pelo número de bits diferentes entre o que seria emitidos pelo estado e os bits recebidos.

Por exemplo, a partir dos bits “00”, a tabela de emissão permite que seja emitido “00” (caso a entrada seja 0) ou 11 (caso a entrada seja 1), além disso, a tabela de transição permite que o estado vá para “00” (caso a entrada seja 0) ou 10 (caso a entrada seja 1). Como o primeiro par recebido foi “00”, temos 2 opções: ficar no estado “00” com um erro de 0, ou ficar no estado 10 com erro de 2, como visto na tabela abaixo:

Estado Inicial	Entrada	Emitidos	Recebidos	Estado Final	Erro
00	0	00	00	00	0
00	1	11	00	10	2

Se o próximo par recebido fosse “11”, teríamos então 4 opções de estado final:

Estado Inicial	Entrada	Emitidos	Recebidos	Estado Final	Erro
00	0	00	11	00	$0 + 2 = 2$
00	1	11	11	10	$0 + 0 = 0$
10	0	10	11	01	$2 + 1 = 3$
10	1	01	11	11	$2 + 1 = 3$

Quando há mais de um caminho para um estado final, como não é o caso do exemplo acima, escolhe-se sempre o com menor erro ou caso eles tenham o mesmo erro escolhe-se um aleatoriamente. Continua-se o processo até o fim. No final dessa etapa pega-se o caminho com menor erro associado e faz-se o caminho inverso obtendo assim, os estados mais prováveis e as transições que ocorreram entre eles. Para achar a entrada original faz-se a emissão dos bits para cada estado que a transição indica e despreza-se os 2 últimos bits.

Como final do procedimento, realiza-se a comparação entre os valores da entrada original com os valores obtidos na decodificação. É verificado então, em quanto a cadeia resultante após as fases do algoritmo se difere da entrada inicial.

## 1.2. Descrição dos problemas e soluções usadas

O desenvolvimento do trabalho codificador/decodificador convolucional usando o algoritmo de Viterbi, cuja descrição consta na seção anterior, foi feita utilizando a linguagem Java e sua estrutura de modelagem baseada em objetos.

Inicialmente, no momento da execução do programa, o usuário irá passar como argumento a cadeia de bits de entrada e o valor de ruído a ser estabelecido nos bits codificados. Sendo que as instruções para compilação e execução do algoritmo constam detalhadas no arquivo `readme.txt`.

Após a execução, no arquivo `Main.java` vai ser feito o tratamento devido aos argumentos recebidos e em seguida em caso de sucesso o encaminhamento para o arquivo `TransicaoEstado.java`, nessa classe irão ocorrer todas as etapas do programa, como codificação, geração do ruído, decodificação e comparação dos resultados finais. A chamada para as fases do algoritmo é feita pela função *processos*.

No arquivo `TransicaoEstado.java`, primeiramente o construtor dessa classe irá construir e popular um Hash Map com os estados 00, 01, 10, e 11 e seus devidos valores de emissão e transição. Em seguida, a partir da função *codificacao*, os bits de entrada recebidos como argumento concatenados com os bits “00” são codificados um a um seguindo as transições de estado e emissões guardadas no Hash Map. O processo de codificação se encerra quando é feita a leitura de todos os bits da entrada, gerando assim uma nova string, sendo essa armazenada na variável do tipo String cujo nome é *bits\_emissao*.

A próxima fase, refere-se a geração de ruído à cadeia de bits *bits\_emissao* resultante da codificação. A função com nome de *estabelece\_ruído* é responsável por fazer o que foi supracitado. Nesse método, a partir do valor de ruído estabelecido é feito um sorteio aleatório para cada bit contido em *bits\_emissao* pela função *Random()*, tal que os valores do ruído e do sorteio pertencem ao conjunto [0, 1]. A cada bit de *bits\_emissao* é feito um teste, onde se o valor aleatório obtido for menor que o ruído o mesmo sofrerá alteração de valor. Se o bit possuía valor igual a ‘0’ passará a ser ‘1’ e vice-versa.

A função *decodifica* simula a criação de uma árvore de transição entre os estados. A árvore começa com o estado “00” sendo a raiz. A partir da raiz, um novo nó é adicionado para cada estado alcançável pelo estado atual através da tabela de transições. O erro é calculado considerando a diferença entre os bits emitidos e recebidos para cada ramo. Os bits emitidos são os bits definidos na Tabela de Transições e Emissões, já os bits recebidos são os pares de bits lidos da sequência codificada após a aplicação do ruído. Ao término da execução dessa função, o resultado será o caminho com o menor erro sobre a árvore, ou seja, retornará a sequência de bits, desprezando os dois últimos bits, mais próxima o possível da entrada original.

Segue abaixo um exemplo de uma árvore de transição para a cadeia de bits 101 e ruído 0.1. Após a codificação e aplicação do ruído temos a cadeia de bits 1110001011. Ao fim da decodificação teremos novamente a sequência 101 demonstrada pelo caminho marcado em vermelho sobre a árvore. Como citado anteriormente, os dois últimos bits serão desprezados, portanto, os dois últimos níveis da árvore não serão mostrados.

El diagrama muestra un árbol de búsqueda de Huffman para la codificación de los bits recibidos 11, 10, 00. El árbol comienza con el estado 00. Las ramas están etiquetadas con 0 y 1. Las ramas resultantes de la codificación de los bits recibidos están resaltadas en amarillo.

Legend:

- Bits emitidos (Green)
- Erro (Red)
- Resultado (Yellow)

Teste com entrada igual a 1000000000000000000000000 e ruído igual 0.25 (25%)

<b>Bits de Entrada :</b>	100000000000000000000000
<b>Ruído:</b>	0.25
<b>Após a Codificação :</b>	11101100 000
<b>Após o Ruído :</b>	11011100101000000000110001011011000010010100010101010 000
<b>Após a Decodificação:</b>	11010000001101000011011000
<b>Número de bits diferentes entre a entrada inicial e a cadeia decodificada:</b>	9

Teste com entrada igual a 10000000000000000000000000 e ruído igual 0.5 (50%)

<b>Bits de Entrada :</b>	100000000000000000000000
<b>Ruído:</b>	0.5
<b>Após a Codificação :</b>	111011000000000000000000000000000000000000 000
<b>Após o Ruído :</b>	000010101100010010111000101011010001010001001101110110 01
<b>Após a Decodificação:</b>	00100000010101101101100110
<b>Número de bits diferentes entre a entrada inicial e a cadeia decodificada:</b>	12

Teste com entrada igual a 1000000000000000000000000 e ruído igual 0.75 (75%)

<b>Bits de Entrada :</b>	100000000000000000000000
<b>Ruído:</b>	0.75
<b>Após a Codificação :</b>	111011000000000000000000000000000000000000 000
<b>Após o Ruído :</b>	00010111111011110111111101111100101100110111101111101 1
<b>Após a Decodificação:</b>	11001001100100100101001100
<b>Número de bits diferentes entre a entrada inicial e a cadeia decodificada:</b>	10

Teste com entrada igual a 1000000000000000000000000 e ruído igual 1 (100%)

[illegible]

Conclui-se então a partir dos testes realizados, que o algoritmo desenvolvido atendeu as requisições impostas na descrição do trabalho, consistindo em um Codificador/Decodificador convolucional utilizando o Algoritmo de Viterbi.