# Implementação do algoritmo de Viterbi para decodificação

**Acácia dos Campos da Terra, João Pedro Winckler Bernardi**

[1]Universidade Federal da Fronteira Sul (UFFS)
Caixa Postal 181 – 89802-112 – Chapecó – SC – Brasil

terra.acacia@gmail.com, jpwbernardi@hotmail.com

***Resumo.*** *Esse documento tem por função descrever o programa desenvolvido cujo objetivo é, dada uma cadeia de bits e uma porcentagem $e$ de erros a ser aplicada sobre ela, realizar a codificação da cadeia, aplicar um ruido, que alterará o estado de $e\%$ dos bits e aplicar o algoritmo de Viterbi para decodificar a cadeia alterada e assim voltar para o estado mais próximo possível do original. Além disso, serão apresentados os resultados obtidos.*

## 1. Introdução

Um código convolucional é um tipo de código corretor de erros. Assim, esse trabalho apresenta um código convolucional codificador e decodificador utilizando o Algoritmo de Viterbi e os resultados obtidos a partir do programa implementado.

Primeiramente, será descrito como é realizada a codificação, seguido do método utilizado para aplicar ruído na cadeia de bits codificada e, por fim, a decodificação que utiliza o Algoritmo de Viterbi. Após a compreensão do programa implementado, serão apresentados os resultados obtidos com o mesmo.

## 2. Entrada do programa

A entrada do programa é dada da seguinte forma:

- um inteiro podendo variar de 0 a 100, que representa a porcentagem de *bits* da cadeia que serão alterados pelo ruído
- uma cadeia de *bits* a ser codificada e decodificada

Exemplo:

```
10        //porcentagem de erros
1001      //cadeia de bits
```

Essa entrada de exemplo será utilizada no decorrer deste documento.

## 3. Codificação

Recebida a entrada, inicia-se o processo de codificação. Para cada *bit*, serão gerados 2 *bits* de saida, porém, devido a mudança de estados que ocorre durante a codificação, esse *bit* acaba influenciando a geração de 3 pares de *bits* de saída. Para usar toda a influencia dos *bits* de entrada, são acrescentados dois *bits*(00) ao final deles. Os estados possíveis são 00, 01, 10, 11 e a codificação e mudança dos estados é dada pela tabela 1.

| | Par de saída | | Próximo estado | |
|---|---|---|---|---|
| **Estado Atual** | **0** | **1** | **0** | **1** |
| **00** | 00 | 11 | 00 | 10 |
| **01** | 11 | 00 | 00 | 10 |
| **10** | 10 | 01 | 01 | 11 |
| **11** | 01 | 10 | 01 | 11 |

**Tabela 1. Tabela de Codificação**

Considerando que o estado inicial é 00, e seguindo a entrada fornecida(1001), concatenando os *bits* 00 no final da sequência, a codificação ocorrerá da seguinte forma: como o primeiro *bit* da sequência é 1, será codificado com 11 e o estado mudará para 10. O segundo *bit* é 0, o que irá gerar os *bits* 10 e o estado irá mudar para 01. O terceiro *bit* é 0, gerando os *bits* 11 e mudando o estado para 00. O quarto *bit* é 1, gerando os *bits* 11 e mudando o estado para 10. O quinto *bit* é 0, são gerados os *bits* 10 e o estado muda para 01. O ultimo bit é 0, gerando 11 e mudando o estado para 00.

Assim, ao final do processo, será obtida a sequência codificada: 11 10 11 11 10 11.

No arquivo *main.cpp*, que contém o código do programa, a codificação é feita pela função *codifica*, a qual é mostrada abaixo:

```
void codifica() {
    //bit é a cadeia da entrada concatenada com 00
    //state é o estado atual
    //saida a cadeia codificada
    string bit = "", state = "00", saida = "";
    for (auto& b: bitin)
        bit += (b + '0');
    bit += "00";
    //cod e prox são maps com os dados da tabela de codificação
    for (auto& b: bit)
        if (b == '0')
            { saida += cod[state].first; state = prox[state].first; }
        else
            { saida += cod[state].second; state = prox[state].second; }
    for (auto& b: saida) bitcod.push_back(b - '0');
}
```

## 4. Ruído

O ruído é responsável por provocar a alteração da sequência codificada. Assim, seja $n$ o número de *bits* informado e $e$ a porcentagem de erro desejado (também informado), $\lceil \frac{n \cdot e}{2} \rceil$ *bits* da entrada codificada serão alterados aleatoriamente.

O ruído aplicado à sequência de *bits* foi gerado a partir de um método aleatório. A função que implementa esta etapa do processo pode ser encontrada no arquivo *main.cpp* e está descrita abaixo.

```
void ruido() {
    int _i, _j = 0, r, i;
    set<int> indices;
    //Adicionamos todos os indices disponíveis num set
    for (i = 0; i < int(bitcod.size()); i++) indices.insert(i);
    int qtderr = ceil(bitcod.size() * (err / 100.0));
    for (auto& b: bitcod) bitnoise.push_back(b);
    //Escolhe qtderr \emph{bits} aleatoriamente para ser alterado
    while (qtderr--) {
        r = rand() % indices.size();
        _i = 0;
        for (auto& i: indices) {
            //Uma vez que um indice foi escolhido, é removido
            //da lista
            if (_i == r) { _j = i; indices.erase(i); break; }
            _i++;
        }
        bitnoise[_j] = !bitnoise[_j];
    }
}
```

## 5. Algoritmo de Viterbi

O algoritmo de Viterbi funciona de forma a determinar todas as possibilidades de decodificação possíveis e escolher a que tem o menor erro. Segue um exemplo para melhor entendimento:

Considerando que a etapa anterior, de aplicar o ruído, gerou a sequência 11 10 11 10 11 11 como resultado, então a sequência irá passar por três etapas, a seguir descritas.

- Primeira etapa
  Como o estado inicial é sempre 00, há somente duas possibilidades a serem seguidas. De acordo com o exemplo que está sendo seguido neste trabalho, o primeiro par de *bits* codificados é 11. No estado 00 (atual), pode ser gerado como resultado 00, quando o *bit* recebido for 0, ou 11, quando o *bit* recebido for 1. Se o *bit* recebido for 0, então o erro por esse estado é igual a 2, pois haverão dois *bits* diferentes do esperado. Se o bit recebido for 1, o erro por esse estado é 0, pois seria gerado exatamente o par que está sendo analizado(11).
- Segunda etapa
  Nesta etapa será analisado o segundo par de *bits*, os quais no exemplo seguido são iguais a 10. Estando no estado 00, os *bits* que podem ser gerados são 00 ou 11. Em ambos os casos, o erro será igual a 1, ou seja há um bit diferente do resultado esperado. Estando no estado 10, podem ser gerados 10 ou 01, sendo o primeiro com erro igual a 0 e o segundo com erro igual a um. Este erro é acumulado individualmente na sequência que está sendo construída.
- Terceira etapa
  O processo se repete nesta etapa, a diferença é que neste momento a quantidade de sequências possíveis vai aumentando exponencialmente. Ao fim do processo, toma-se a sequência que acumulou a menor quantidade de erro e, percorrendo

o caminho inverso, será obtida a melhor sequência de *bits* muito similar ou até mesmo igual a que foi informada na entrada anteriormente.

A forma de implementação foi através de uma programação dinâmica(PD), pois existem muitas possibilidades de caminhos a serem seguidos para obtenção da sequência. A PD tem por função tratar estes casos através de uma memorização dos caminhos já calculados anteriormente. A implementação em forma de código da PD está exibida abaixo. A explicação da mesma se encontra logo após.

```
int Viterbi(int i, int j) {
  //Se o estado da PD já foi calculado, retorna o valor.
  if(pd[i][j] != -1) return pd[i][j];
  string par;
  int err0 = 0, err1 = 0, pai0 = 0, pai1 = 0;
  //Se estamos no primeiro par de bits e o estado atual da sequencia
  //não for 00 ou 10, não é uma sequência válida e o erro é definido
  //como INFINITO para ela
  if (i == 2 && j != 0 && j != 2) return INF;
  par = getbitn(i - 2); par += getbitn(i - 1);
  //Para o primeiro par de bit, analisamos as unicas duas opções
  //possíveis
  if (i == 2)
    return pd[i][j] = min(calcerror(par, "00"), calcerror(par, "11"));
  //Dependendo do estado atual da PD, um caso deve ser verificado,
  //conforme a tabela de codificação
  //É definido o pai do estado para a reconstrução do caminho
  switch (j) {
  case 0:
    err0 = calcerror(par, "00") + Viterbi(i - 2, 0);
    err1 = calcerror(par, "11") + Viterbi(i - 2, 1);
    pai0 = 0; pai1 = 1;
    break;
  case 1:
    err0 = calcerror(par, "10") + Viterbi(i - 2, 2);
    err1 = calcerror(par, "00") + Viterbi(i - 2, 3);
    pai0 = 2; pai1 = 3;
    break;
  case 2:
    err0 = calcerror(par, "11") + Viterbi(i - 2, 0);
    err1 = calcerror(par, "00") + Viterbi(i - 2, 1);
    pai0 = 0; pai1 = 1;
    break;
  case 3:
    err0 = calcerror(par, "01") + Viterbi(i - 2, 2);
    err1 = calcerror(par, "10") + Viterbi(i - 2, 3);
    pai0 = 2; pai1 = 3;
  }
  if (err0 < err1) pai[i][j] = pai0;
```

```
  else pai[i][j] = pai1;
  return pd[i][j] = min(err0, err1);
}
```

Na PD, o inteiro $i$ representa a quantidade de *bits* que não foram decodificados ainda e o inteiro $j$ representa o estado atual do caminho. Assim, o estado $S_{i,j}$ representa a menor quantidade de erros que é possível ser obtido na sequência até o *i-ésimo bit*, terminando no estado $j$. A variável $i$ pode variar de 0 ao tamanho da sequência com ruído aplicado e a variável $j$ pode variar de 0 a 4, onde cada valor representa, respectivamente, os estados 00, 01, 10 e 11.

A partir dessas informações, é feita uma análise: estando no estado $j$, quais estados poderiam alcança-lo. Analisa-se também os *bits* gerados por esses estados para alcançar $j$. Então, somamos o menor erro dentre as opções disponíveis com o erro da melhor sequência até o par de *bits* anterior com o estado que alcança $j$ e, assim, conseguimos calcular o estado $S_{i,j}$ da PD.

A base da PD é o caso $i = 2$, onde será feita a análise do primeiro par de *bits* da sequência. Se o estado $j$ for diferente de 00 ou 10, a sequência é inválida, pois o estado inicial da codificação é 00 e não seria possível a sequência alcançar $j$. Como não há mais *bits* a serem analisados, basta calcular o erro desse estado.

É gravado o estado anterior da sequência para cada estado da PD, através disso é possível reconstruir o caminho.

## 6. Resultado

Nesta seção do documento serão apresentados exemplos de execução e os resultados obtidos para os mesmos.

Para a sequência de *bits* de entrada 1001, considerando 25% de ruído, a sequência decodificada obtida foi idêntica à originalmente fornecida.

Resultado satisfatório foi obtido também para a sequência 001110, onde foi considerado 17% de ruído. Na tabela 2 estão exibidos outros exemplos para diferentes porcentagens de ruído, cada exemplo foi executado 20 vezes, e, assim, calculou-se o erro médio.

| Entrada | % | Entrada Codificada | Erro Médio |
|---|---|---|---|
| 001110 | 5 | 0000110110011100 | 0 |
| | 15 | | 0.9 |
| | 25 | | 0.65 |
| | 50 | | 2.15 |
| 11010101000010100001 | 5 | 1101010010001000101100001110001010000111011 | 1.05 |
| | 15 | | 4.15 |
| | 25 | | 7.35 |
| | 50 | | 9.65 |

**Tabela 2. Resultados**

## 7. Conclusão

O trabalho consiste de um programa em que, dado um valor $x$ que representa a porcentagem de erro aplicado, e uma sequência de *bits*, codifica-se essa sequência, aplica-se ruído a codificação(alterando x$\%$ *bits*) e, por fim, realiza-se a decodificação da sequência codificada com ruído através do algoritmo de Viterbi.

O maior problema enfrentado foi como computar todas as possibilidades de caminho no momento da decodificação de uma forma eficiente. Por isso, o Algoritmo de Viterbi foi implementado fazendo o uso de programação dinâmica, de forma a evitar recomputação de valores. Essa abordagem se mostrou bastante efetiva e vantajosa.

Não só o programa computa a sequência de forma bastante rápida, como também gera uma sequência com poucos erros. Com a taxa do ruído em 15$\%$, frequentemente é obtida a sequência original como resultado e, até mesmo com taxa do ruído em 50$\%$ houveram situações em que a sequência gerada foi exatamente a mesma da fornecida na entrada em menos de 20 tentativas. Isso mostra a eficácia do código e da implementação feita.