Análise de complexidade de um algoritmo para visualização da árvore de frequência da codificação de Huffman

Fernanda G. Islas Martínez[1]

[1]Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Av. Ipiranga, 6681 – Partenon – Porto Alegre – RS – Brasil

Resumo. *A codificação de Huffman é um dos primeiros algoritmos para compressão de dados. A codificação utiliza códigos de diferentes comprimentos, atribuindo códigos mais curtos para caracteres de maior frequência. Para calcular a frequência de caracteres é utilizada uma árvore binária. O objetivo deste trabalho é determinar a complexidade de um algoritmo que permite a visualização da árvore utilizada na compressão. A conclusão é de que a complexidade depende do valor do tamanho da arvore e pode ser descrita como T(n)=O(n*log2(n)).*

1. Introdução

Os Codigos Huffman são códigos de compressão de datos resultado de excelentes trabalhos do Prof. David A Huffman (1925-1999) .Os códigos Huffman exploram a forma da mensagem para dar boas taxas de compressão. O esquema de codificação Huffman é um método de codificação com comprimento variável. Isto quer dizer que o código para um símbolo depende da frequência de ocorrência do símbolo referido. A codificação de Huffman é classificada em dois grupos diferentes, Huffman estático e Huffman adaptável. Na codificação Huffman estática, as frequências dos símbolos são conhecidos antecipadamente, já se sabe qual o símbolo que irá ocorrer e quantas vezes.

Os árvores Huffman são estritamente árvores binários, em que todos os nós, exceto os nós folha têm ambos as crianças direita e esquerda.

O objetivo deste trabalho é determinar a complexidade de um algoritmo que permite a visualização da árvore utilizada na compressão. A tarefa proposta apresenta três etapas: (a) localizar ou desenvolver uma implementação do algoritmo de Huffman, (b) exportar a árvore utilizada pela implementação em um arquivo texto no formato utilizado pelo sistema graphviz e (c) determinar a complexidade do algoritmo utilizado na exportação.

A conclusão é de que a complexidade depende do valor de (indicar valor de entrada) e pode ser descrita como (classe de complexidade).

2. Metodologia

O inicio da pesquisa foi na internet, procurando palavras claves como 'Huffman Code' e 'Huffman algorithm' do inicio para a compressão do algoritmo e depois procurando uma implementação em linguaje de programação Java. A implementação foi localizada no GitHub, usando a pesquisa por repositório com a expressão huffman, com filtro para Java e ordenação pelo número de bifurcações. O projeto selecionado foi [Wu 2014].

3. Implementação da codificação de Huffman

A implementação obtida de o site GitHub, na linguagem Java é uma implementação do HuffMan Algorithm que tem integrada uma interface gráfica para o uso de usuário, com opções de "Compressão"e "Descompressão"de arquivos.

4. Exportação no formato graphviz

Para a implementação do algoritmo de Huffman agrego-se um método para criar um arquivo com formato para o sistema Graphviz.

```
// Create a Huffman tree based on the frequency table
this.codeTree = new HuffmanTree(countObj);

System.out.println("\nHuffman Tree.sintaxGraphviz(): \n\n" + this.codeTree.syntaxGraphviz());
```

Figura 1. syntaxGraphvizM

4.1. Algoritmo

O metodo utiliza a arvore binaria gerada pelo algoritmo de Huffman, a exportação e feita por dois métodos. O primero *"syntaxGraphviz"* (Figura2) imprime o inicio do formato e chama ao seguinte metodo *"syntaxGraphviz0"* (Figura3) o qual e recursivo e va percorrendo o arvore desde a raiz.

O formato utilizado nessa implementação só contem a declaração de um digraph "Grafo_Huffman", o tamanho dos nós e a forma do nó, além a chamada a *"syntaxGraphviz0"*.

```
public String syntaxGraphviz(){

    return "digraph Grafo_Huffman{\n" + "size=\"8\"\n" + "node [shape = circle];\n"
            + syntaxGraphviz0(root) + "}" ;
  }
```

Figura 2. syntaxGraphviz

SyntaxGraphviz0, só tem o nó raiz como parâmetro, o primeiro que face o método é criar um *String* "nodos"no qual armazena-se toda a sintaxes. Se o nó o que passa-se como parâmetro ao método e diferente de *null* valida se o nó atual tem filhos para saber se fosse folha o não.

Se o nó não é folha então agrega ao *String* o identificador do nó *"root.id"* e chama *labelGraphviz()* que imprime a frequência do nó. O seguinte e imprimir as relações do nó, pela esquerda e pela direita, assim como o valor da aresta (0 ou 1).

Se o nó atual fosse uma folha então imprime o identificador *"root.id"* e chama *labelGraphviz* que imprime nesse caso a frequência do nó assim como ao carácter da folha.

O método retorna o *String "nodos"* e faz a chamada recursiva pela esquerda e logo à direita.

```
public static String syntaxGraphviz0(HuffNode root){
    String nodos= "";

    if(root != null ){
        if (root.left != null) {           //O no nao e folha, imprime a frequencia da folha
            //if (root.value!= ERROR && (int)root.value != INCOMPLETE_CODE) {
                nodos += root.id + labelGraphviz(root, false)+
                        root.id +" -> "+ root.left.id + labelGraphviz(true)
                        + root.id +" -> "+ root.right.id + labelGraphviz(false);

        }
        else
                nodos += root.id + labelGraphviz(root, true);

        return nodos + syntaxGraphviz0(root.left)+ syntaxGraphviz0(root.right);
    }

    else
        return nodos;
}
```

Figura 3. syntaxGraphviz0

labelGraphviz(boolean izq) (Figura4) quando a função é chamada, dentro dela vai declarar um *String* vazio donde se o parâmetro booleano *izq* é verdadeiro, quer dizer que a relação do nó vai à esquerda e o label da aresta é 0. No caso contrario o parâmetro foi falso, então a relação vai à direita e o label da aresta é 1.

```
public static String labelGraphviz(boolean izq){
    String label = "";

    if(izq == true)
        return label+= "[ label = 0 ];\n";

    return label+= "[ label = 1 ];\n";
}
```

Figura 4. labelGraphviz()

labelGraphviz(HuffNode root, boolean caracter) (Figura5) e uma sobrecarga do método, quando a função e chamada, dentro dela também vai declarar um *String* vazio donde se o parâmetro booleano carácter é verdadeiro, quer dizer que o nó é folha então vai escrever o carácter e frequência do nó. La também valida-se que o *root.value* esteja acima de 31 já que no código ASCII todos os valores abaixo do 32 são caracteres de controle, assim como acima de 254 que em nosso arvore não precisamos.

No caso contrario o parâmetro booleano foi falso, então o nó não é folha e unicamente precisa de escrever a frequência do nó.

```
public static String labelGraphviz(HuffNode root, boolean caracter){
    String label = " ";

    if (caracter && root.value>31 && root.value<254) {
        return label+= "[ label =\""+ (char)root.value +", "+ (int)root.weight+" \"];\n";
    }
    else
        return label+= "[ label ="+ root.weight+" ];\n";

}
```

Figura 5. labelGraphviz()

4.2. Alterações na implementação

Um problema encontrado na exportação do formato foi o identificador do nó, num principio usava o mesmo valor da frequência como identificador, mas a maioria das vezes tem nós com a mesma frequência, no formato *Graphviz*junta só, embora não sejam o mesmo nó.

Para evitar usar a frequencia do no, a solução foi criar na classe *HuffmanNode* um contador e um novo atributo para o construtor do objeto *HuffmanNode* mesmo que vai usar o contador que vai aumentar com cada nó criado. Assim os novos nós sempre vai ter um identificador diferente que vai ser usado unicamente na exportação *Graphviz*.

```
public class HuffNode implements Comparable<HuffNode> {
        public int value;// Node value
        public int weight;// weight
        HuffNode left, right;// left/right child node
        HuffNode parent;// parent node
        public static int contador = 0; //ID node
        public final int id;

        HuffNode(int v, int w, HuffNode lchild, HuffNode rchild, HuffNode pt) {
                value = v;
                weight = w;
                left = lchild;
                right = rchild;
                parent = pt;
                id = contador++;
        }
```

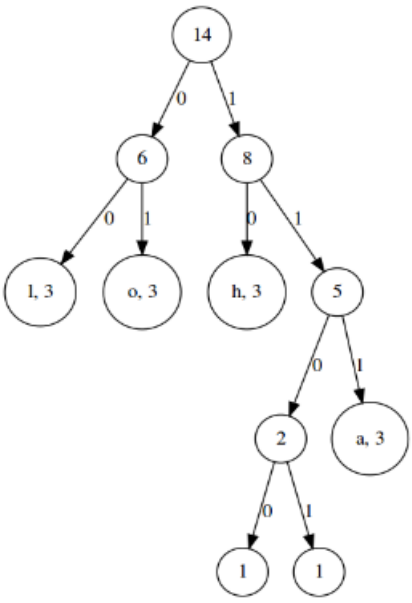
Figura 6. HuffmanNode

4.3. Testes

Para comprovar o correto funcionamento da implementação e do método **syntaxGraphviz** foram criados alguns testes de compressão, descompressão e formato *graphviz*.

4.3.1. Teste#1(Figura 7) *"holaholahola"*

No primeiro teste, provou que as folhas tuvesem o caracter e a frequência, la observamos que o arvore tem dois nós folhas sem carácter, inseridos pelo algoritmo Huffman são o inicio (ASCII#255,) e fim da leitura do arquivo(ASCII #10,
).




```
digraph Grafo_Huffman{
size="8"
node [shape = circle];
23 [ label =14 ];
23 -> 21[ label = 0 ];
23 -> 22[ label = 1 ];
21 [ label =6 ];
21 -> 16[ label = 0 ];
21 -> 17[ label = 1 ];
16 [ label ="l, 3 "];
17 [ label ="o, 3 "];
22 [ label =8 ];
22 -> 15[ label = 0 ];
22 -> 20[ label = 1 ];
15 [ label ="h, 3 "];
20 [ label =5 ];
20 -> 19[ label = 0 ];
20 -> 14[ label = 1 ];
19 [ label =2 ];
19 -> 13[ label = 0 ];
19 -> 18[ label = 1 ];
13 [ label =1 ];
18 [ label =1 ];
14 [ label ="a, 3 "];
}
```

Figura 7. Teste#1

4.3.2. Teste#2(Figura 8)*"aaaaaeeeeiiioou"*

No segundo teste foi para provar as frequências e arestas corretas.

```
digraph Grafo_Huffman{
size="8"
node [shape = circle];
12 [ label =17 ];
12 -> 10[ label = 0 ];
12 -> 11[ label = 1 ];
10 [ label =7 ];
10 -> 3[ label = 0 ];
10 -> 2[ label = 1 ];
3 [ label ="i, 3 "];
2 [ label ="e, 4 "];
11 [ label =10 ];
11 -> 9[ label = 0 ];
11 -> 1[ label = 1 ];
9 [ label =5 ];
9 -> 7[ label = 0 ];
9 -> 8[ label = 1 ];
7 [ label =2 ];
7 -> 0[ label = 0 ];
7 -> 6[ label = 1 ];
0 [ label =1 ];
6 [ label =1 ];
8 [ label =3 ];
8 -> 5[ label = 0 ];
8 -> 4[ label = 1 ];
5 [ label ="u, 1 "];
4 [ label ="o, 2 "];
1 [ label ="a, 5 "];
}
```

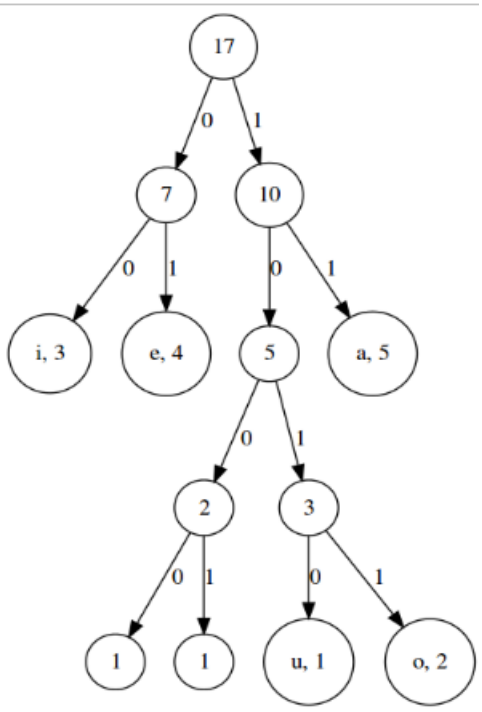


Figura 8. Teste#2

4.3.3. Teste#3(Figura 9)*Texto-Imagen*

No terceiro teste foi para provar a capacidade, se inseriu um paragrafo e logo uma imagem.

4.4. Arquivo .SGV

Ao terminar a exportação do formato, o programa cria um arquivo *.sgv* donde vai armazenar a sintaxes gerada.

A implementação do método foi feita na classe *HuffmanTreeOutputStream*. (Figura.10)

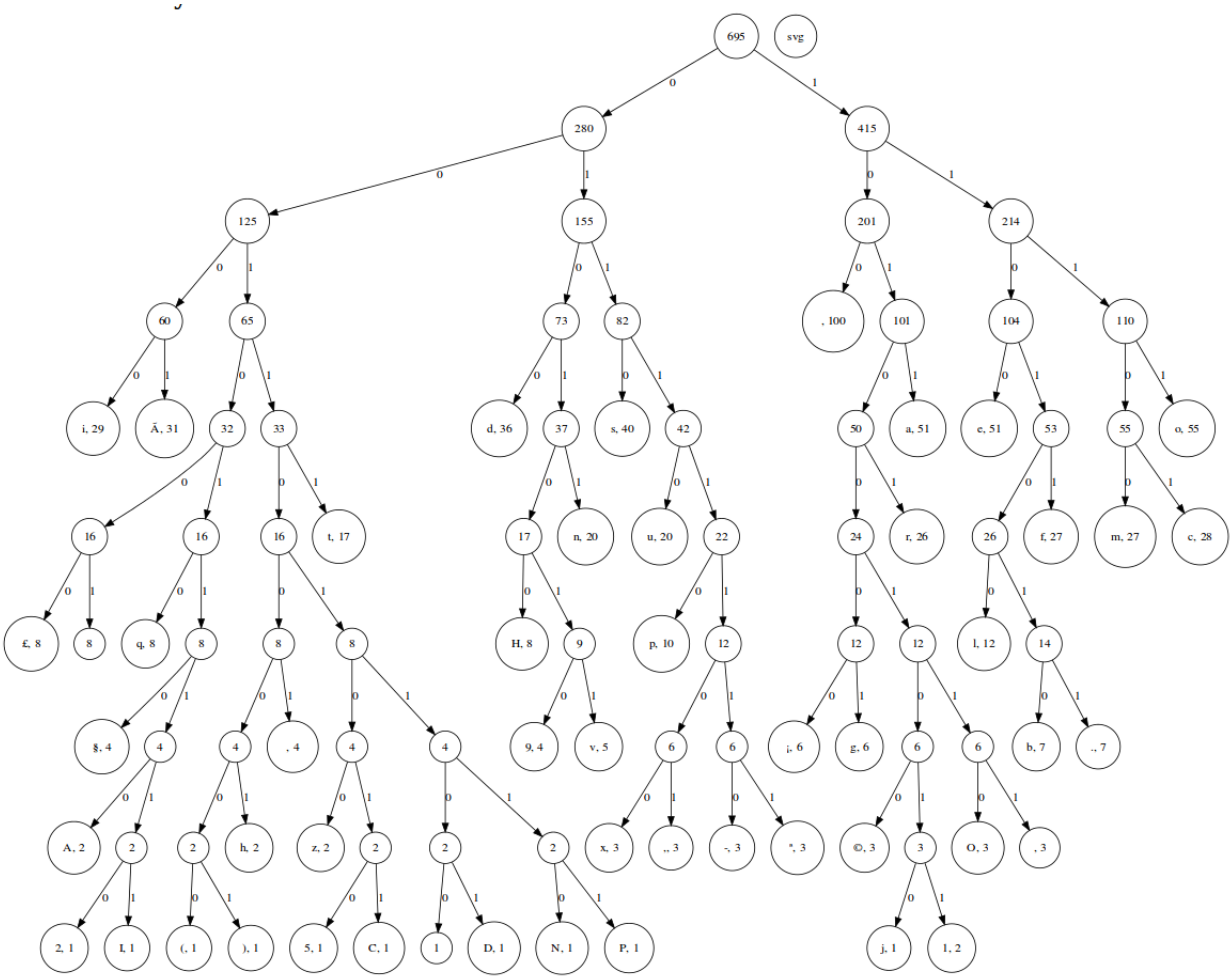


Figura 9. Teste#2

```
public void graphvizFile(String graphviz) throws IOException{
    //String pathraiz = file.getPath();
    //String name = file.getName();
    String nomesgv = ruta + HuffViewer.SXGRAPH_SUFFIX;
    ManejadordeArchivos ma = new ManejadordeArchivos();
    //final File newsgvFile = new File(nomesgv);
    ma.agregaContenidoArchivo(nomesgv, this.codeTree.syntaxGraphviz());
}
```

Figura 10. graphvizFile(String ruta, String name)

4.5. Análise

Para determinar a classe de complexidade, utilizou-se a análise do código do algoritmo. o algoritmo utiliza o corrimento completo da arvore para fazer a exportação do formato *Graphviz*, num percorrido só, se faz tudo sintaxes. A complexidade do primeiro método *syntaxGrahpviz()* e linear T(n)=5 já que é chamada uma vez só.

syntaxGrahpviz0() é o método que implementa o algoritmo, nele se faz a chamada recursiva do método, e percorre o arvore binário Huffman. A complexidade assim é logarítmica T(n)=n*log2(n)

5. Conclusão

Para realizar esse trabalho se utilizo uma implementação encontrada no internet, a qual primeiro houvo que estudar e compreender a estrutura do código. O primeiro passo e o mais difícil, foi encontrar a parte donde o arvore é criado, a implementação cria um arvore pela frequência o qual precisava, uma vez encontrado, primeiro tente imprimindo tudo o arvore com uma função recursiva, tive como resultado só números.

Na classe **HuffmanNode** os atributos frequência(*root.weight*) e carácter(*root.value*) são tipo inteiros, então unicamente fiz *cast* para os caracteres.

Ao fazer a arvore deu conta de que a implementação insira dois caracteres mais à arvore, com os teste verifiquei que um é o inicio da leitura do arquivo e o outro é um salto de línea que marca o fim da leitura.

Um problema encontrado na exportação do formato foi o identificador do nó, num principio usava o mesmo valor da frequência como identificador, mas a maioria das vezes tem nós com a mesma frequência, no formato *Graphviz* junta só, embora não sejam o mesmo nó. A solução foi criar na classe HuffmanNode outro identificador para usar ele na exportação.

No sistema *Graphviz* aceita vários formatos para gerar a arvore, mas como alguns testes foram com imagens, a exportação gera muitos caracteres, um deles é '"', na hora de passar ao formato tem confissões com a sintaxes, é precisão evitar eles caracteres ou trocar eles por outro.

Referências

Wu, E. (2014). Reference huffman coding. https://github.com/entingwu/HuffmanCoding.