Huffman

Análise experimental

Thiago Basso
Acadêmico do Mestrado Profissional em Computação Aplicada
Universidade Federal de Mato Grosso do Sul, UFMS
Campo Grande - MS
thiago.basso@gmail.com

***Resumo* — Artigo apresentado à disciplina de Análise de Algoritmo, que trata da elaboração de experimentos relacionados a algoritmos de compactação/descompactação construída utilizando-se a ferramenta Octave, e implementada seguindo a estratégia de códigos de prefixos de Huffman.**

Palavras-chave* — *Huffman; compactação; codificação; compressão; código de prefixos

I. INTRODUÇÃO

Compactadores são softwares que implementam uma representação mais eficiente dos caracteres de um arquivo, de modo que seu tamanho total seja reduzido. Existe uma gama enorme de ferramentas comerciais destinadas a esse propósito. Algumas possuem códigos proprietários, ao passo que outras utilizam-se da técnica de código de prefixos de Huffman, pelo fato da mesma não ser patenteada e se mostrar bastante eficiente [5]. O presente artigo faz uma breve descrição da técnica de Huffman e em seguida apresenta os procedimentos realizados para experimentação da mesma. Os resultados coletados são comentados e por fim é feita uma breve conclusão acerca do tema.

II. DESCRIÇÃO DO MÉTODO

No decorrer dessa seção, será apresentado os detalhes do algoritmo de Huffman, seu funcionamento será explicado e exposto um exemplo de sua execução. Será ainda discutido a estratégia de formulação do algoritmo, chamada de Algoritmos Gulosos, e sua complexidade.

A. Códigos de Huffman

David Albert Huffman, Bacharel em Engenharia Elétrica e entusiasta da Ciência da Computação, é a pessoa por trás da técnica de compressão, sem perda de dados, denominada Códigos de Huffman. A técnica foi formulada enquanto estudante de doutorado no MIT – *Massachusetts Institute of Technology*, e apresentada no artigo: *A Method for the Construction of Minimum-Redundancy Codes,* em 1952 [2] [3] [4].

Suponha um arquivo de dados contendo 100mil caracteres. Esse arquivo é composto por apenas seis caracteres: a, b, c, d, e, f; que se repetem a frequências de 45mil, 13mil, 12mil, 16mil, 9mil, e 5mil respectivamente. Existem diversas maneiras de projetar um código de caracteres binário que represente cada um dos caracteres, por exemplo, pode-se usar a formatação ASCII[1] ou ainda um código de comprimento fixo de 3 bits para representar cada um dos seis caracteres ($2_{(\text{base binária})}\text{^}3_{(\text{bits})} = 8$ caracteres, mínimo para se representar seis caracteres). Com esse método é possível codificar todo o arquivo utilizando 300mil bits [1] (em ASCII seriam 800mil bits).

Considerando a *string* "aabbcdddef". Utilizando a codificação ASCII (8bits por caractere), representada na tabela 1, sua escrita binária seria: 0110 0001 0110 0001 0110 0010 0110 0010 0110 0011 0110 0100 0110 0100 0110 0100 0110 0101 0110 0110; totalizando 80bits. Utilizando a codificação fixa de 3bits dada pela tabela 2, sua escrita binária seria: 000 000 001 001 010 011 011 100 101; totalizando 30bits. Em termos de compressão a codificação de 3bits leva vantagem, com uma economia de 50bits em relação a representação em ASCII. Utilizando a técnica de Huffman veremos que é possível fazer algo ainda melhor.

Tabela 1 - Codificação ASCII [3]

Caractere	ASCII	Binário – Palavra de código
a	97	0110 0001
b	98	0110 0010
c	99	0110 0011
d	100	0110 0100
e	101	0110 0101
f	102	0110 0110

Tabela 2 - Codificação Fixa de 3bits [1]

Caractere	Binário – Palavra de código
a	000
b	001
c	010
d	011
e	100
f	101

[1] ASCII – American Standard Code for Information Interchange (Código Padrão Americano para o Intercâmbio de Informação) permite codificar um conjunto de 128 sinais (95 gráficos e 33 de controle), utilizando 7bits em 1byte. O bit não utilizado pode ser utilizado e formas diferentes em cada padrão de codificação (UTF-8, ANSI, etc.) [3] [4].

A técnica de Huffman prevê códigos de comprimento variável, que funciona de maneira mais econômica que a técnica de comprimento fixo. A ideia é que palavras de código curtas são atribuídas aos caracteres mais frequentes, ao passo que, palavras de códigos maiores são atribuídas a caracteres pouco frequentes [1] [4]. Com base no exemplo da *string* "aabbcdddef" e levando em conta a tabela de código de comprimento variável apresentada na tabela 3, a escrita em binário seria: 00 00 111 111 110 10 10 10 011 010; totalizando 25bits. Uma economia de 5bits em relação à codificação fixa.

Tabela 3 - Codificação de Tamanho Variável

Caractere	Binário – Palavra de código
a	00
b	111
c	110
d	10
e	011
f	010

B. Códigos de Prefixo

Um código de prefixo é aquele no qual nenhuma palavra de código é prefixo de uma outra palavra de código [1]. No exemplo da tabela 3, podemos ver que os caracteres 'a' e 'd', que possuem palavra de código de 2bits (os demais 3bits), não são prefixos de nenhuma outra palavra de código. Sendo assim, o código de prefixo sempre consegue a compressão de dados ótima para qualquer sequência de caracteres [1].

Utilizando a codificação de prefixos a tarefa de decodificação é simplificada, pois nenhuma palavra de código será prefixo de outra, e o processo flui simplesmente identificando uma palavra de código inicial, traduzindo-a de volta para o caractere original e repetindo o processo de decodificação para o restante do arquivo [1]. No nosso exemplo, a cadeia 000011111111010101010011010 decodificada seria analisada da seguinte forma: 00 – 00 – 111 – 111 – 110 – 10 – 10 – 10 – 011 – 010, que é decodificada como 'aabbcdddef' (ver tabela 3).

Para representação do código de prefixo, de modo a facilitar a decodificação, o mesmo pode ser representado por uma árvore binária, cujas folhas são os caracteres. Desta forma, cada palavra código será o caminho simples da raiz até a folha, ao passo que 0 significa: "vá para o filho da esquerda" e 1 significa: "vá para o filho da direita" [1], a árvore montada para o exemplo é apresentada na figura 1.

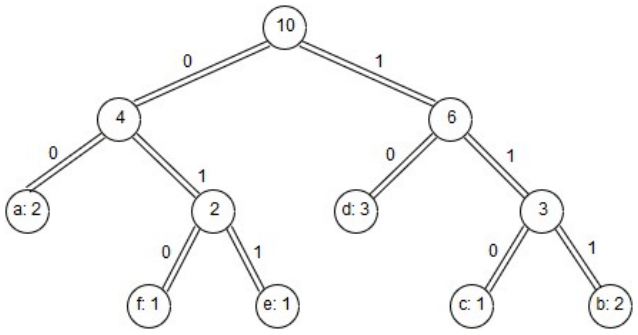

Figura 1 - Árvore binária para o exemplo dado

Uma árvore binária cheia representa um código ótimo para um arquivo. Conforme podemos ver na figura 2, a árvore para o código de comprimento fixo não é ótima, pois o nó interno 14 possuí apenas um filho, ao passo que a árvore para o código de comprimento variável da figura 1 é ótima. Para ser considerada uma árvore binária cheia (código ótimo), todo nó que não é folha deve ter dois filhos [1].

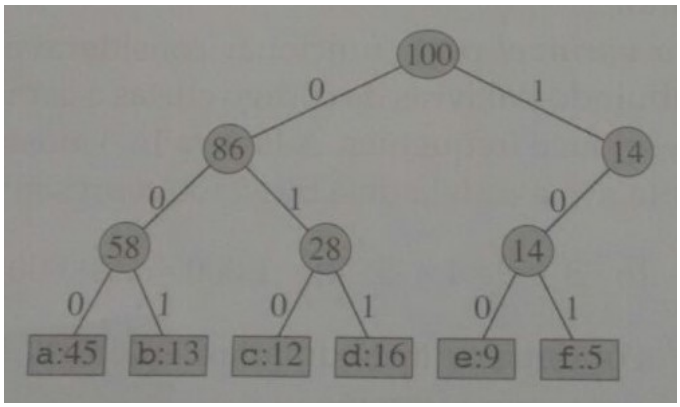

Figura 2 - Árvore binária correspondente ao código de comprimento fixo [1]

C. Construção de Códigos de prefixos ótimo – O algoritmo de Huffman

Huffman criou um método, utilizando algoritmo guloso, capaz de produzir um código de prefixo ótimo. A estratégia gulosa, consiste em realizar a escolha que parecer ser a melhor no momento (localmente ótima), na esperança de que essa escolha leve a uma solução ótima global. Nem sempre essa heurística produz a solução ótima para determinado problema, porém, ela é verdadeira para o algoritmo de Huffman, conforme provado em Cormen et. al. em [1].

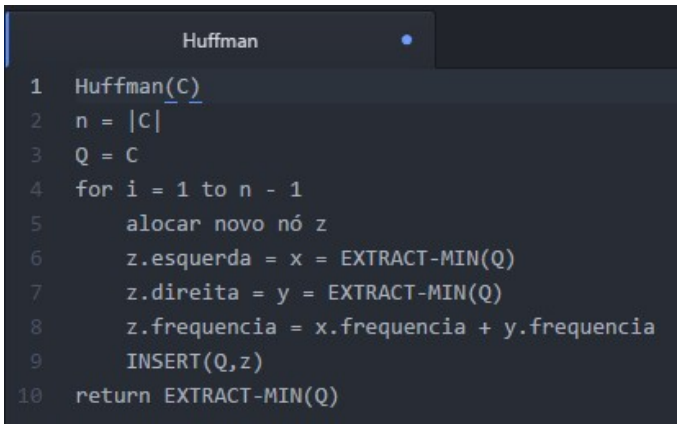

Figura 3 - Algoritmo de Huffman [1]

No pseudocódigo da figura 3, C representa o conjunto de n caracteres em que cada um desses elementos é representado por um objeto com atributo 'frequencia', que representa a frequência com que o caractere aparece no arquivo. n recebe a quantidade inicial de elementos em C (linha 2), ou seja, a quantidade de folhas que terá a árvore. A fila de prioridade mínima Q recebe os objetos de C na linha 3, lembrando que a chave da fila de prioridades é o atributo 'frequencia'. Nas linhas de 4 a 9 a árvore é construída de baixo para cima,

sempre intercalando os dois objetos de menor frequência. O resultado dessa intercalação é um novo objeto, cuja frequência é a soma dos dois, que se torna a raiz temporária dos dois objetos originais, onde o objeto de menor frequência será seu filho da esquerda e o de maior frequência seu filho da direita. O objeto resultante é inserido na fila de prioridades, e participará (será passível de escolha) da próxima interação do *loop for*. Quando o *loop* termina sua interação existirá apenas um objeto na fila de prioridades que corresponde a raiz da árvore (linha 10).

Considere o exemplo de uma fila de prioridades extraída da tabela 3. Nas figuras 4 e 5 é possível acompanhar a movimentação da fila de prioridades Q. Em (a) temos o conjunto inicial de objetos que compõem a fila. Em (b) ocorreu a primeira intercalação entre os objetos de menor frequência (caracteres 'f' e 'e'), a frequência deste objeto é composta pela soma da frequência dos filhos e é inserido em sua posição correta na fila de prioridades, esses passos são repetidos até (f), quando é obtida a árvore final.

Tabela 3 - Caracteres e suas respectivas frequências [1]

	a	**b**	**c**	**d**	**e**	**f**
Frequência	45	13	12	16	9	5

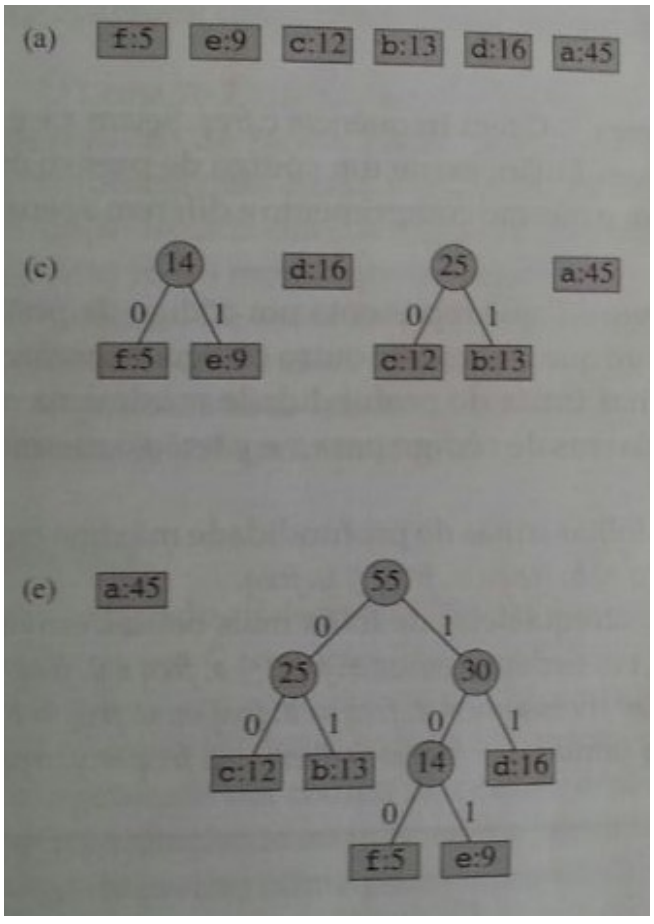

Figura 4 - Huffman passo a passo, etapas a, c, e [1]

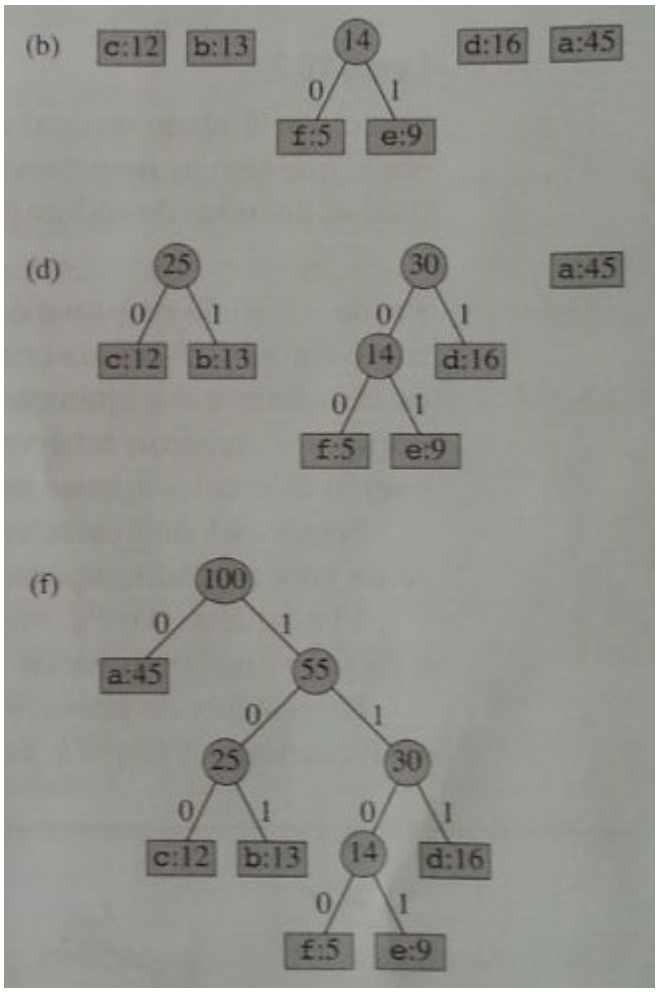

Figura 5 - Huffman passo a passo, etapas b, d, f [1]

Supondo Q implementado como um *heap* de mínimo binário, a complexidade final do algoritmo é dada pelo tempo O(n lg n). Cada operação do laço de repetição contribui com o tempo O(lg n), que corresponde ao tempo de cada operação no heap, de modo que seu tempo de execução é dado por O(n lg n). Sendo o tempo dominante sobre os demais passos do algoritmo, compondo assim o tempo final [1].

III. METODOLOGIA EXPERIMENTAL

O presente trabalho compõe atividade solicitada no estudo da disciplina de Experimentação Algorítmica. Fora solicitada a implementação de um algoritmo compactador/descompactador utilizando a estratégia de código de prefixos de Huffman.

Para implementação das rotinas necessárias fora utilizada a ferramenta GNU Octave, versão 4.0.3. Esta ferramenta foi a indicada para a execução durante a disciplina, devido a sua facilidade de plotar gráficos e executar atividades propostas à experimentação de algoritmos. O ambiente para execução dos experimentos é uma máquina rodando Windows 10 Home – 64bits, cuja configuração é: processador Intel Core i5 @ 2.40GHz, com 4GB de memória RAM.

Procurou-se construir um código com clareza didática em detrimento à performance. Sendo assim, o projeto é composto por diversos módulos organizados da seguinte forma:

- Huffman.m: contempla as rotinas relacionadas ao algoritmo de Huffman necessárias para a criação da árvore de prefixos que é o estágio inicial tanto para as rotinas de compactação e descompactação.
- Codificar.m: organiza rotinas necessárias para codificar uma *string* de entrada. Basicamente a árvore de prefixos é percorrida em pré ordem para criar a tabela de códigos e então a *string* de entrada é codificada.
- Compactador.m: possui as rotinas básicas necessárias ao funcionamento de um compactador. Gerencia a abertura e extração do arquivo texto original, monta a tabela de símbolos (caracteres) e frequências e realiza chamadas aos procedimentos presentes em Huffman.m e Codificar.m para codificação do texto. Apresenta ainda rotina para tratar o arquivo codificado de saída.
- Decodificar.m: rotina única que recebe como entrada a árvore de prefixos e uma cadeia de caracteres codificados. É responsável por percorrer a árvore de prefixos e retornar o texto original.
- Descompactador.m: semelhante ao Compactador.m, porém suas rotinas são voltadas ao funcionamento de um descompactador. Gerencia a abertura do arquivo codificado, extraindo a tabela de símbolos (caracteres) e frequências para montar a árvore de prefixos (via Huffman.m) e também do texto codificado. Realiza chamada a Decodificar.m para retornar ao texto original e então decodificar o arquivo, salvando-o novamente em ASCII.
- Main.m: script utilizado para experimentação, sua rotina é uma série de testes que visam extrair tempos de execução e taxa de compactação para arquivos de diversos tamanhos. Também plota gráficos com análise dos dados.

Os testes foram conduzidos executando o script padrão Main.m. Outros experimentos podem ser realizados de posse da implementação acima expostas, melhores detalhes de como utilizar as rotinas presentes no programa podem ser obtidos lendo o arquivo de README.txt. Os resultados obtidos são discutidos a seguir.

IV. Resultados e Discussões

Os experimentos solicitados mediram o tempo de execução da compactação, tempo de execução da descompactação e a taxa de compactação apresentada pela estratégia de Huffman. Foram testados quatro arquivos com os tamanhos: 50bytes, 500bytes, 20000bytes e 5000bytes, esses arquivos foram gerados por uma ferramenta web que gera textos aleatórios chamada Lorem Ipsum[2], e salvos no formato txt. Os tamanhos utilizados nos cálculos foram extraídos após a leitura do arquivo, ou antes da escrita em arquivo (não foi levado em conta o espaço ocupado por cabeçalho, símbolos e frequências), sendo necessária algumas conversões para representação dos tipos. Por exemplo, a *string* codificada pela árvore de prefixos é um vetor de caracteres (0's e 1's), sendo assim, cada posição do vetor representa 1bit, para o cálculo da taxa de compactação o tamanho é dividido por oito para representar sua conversão em byte e assim ser da mesma grandeza.

Na figura 6 é exibido um resumo dos experimentos realizados. Os tempos da descompactação são bem menores que a compactação, o que já era esperado, pois basta percorrer a árvore de prefixos da raiz, utilizando a orientação dada pelo código, até chegar a folha. Na figura 7 e 8 são exibidos gráficos em linha contendo os tempos gastos para compactação e descompactação de cada um dos arquivos, respectivamente. Podemos observar que o tempo gasto cresce linearmente de acordo com o tamanho do arquivo. Com relação a taxa de compactação, ela pode sofrer variações dependendo da frequência dos caracteres contidos no arquivo, no caso a variação para os experimentos propostos ficou em torno de 46% a 51%, como pode ser visto na figura 9.

```
----------/(arquivo pequeno)/----------

Testando uma entrada pequena de 50 bytes
Tamanho do arquivo compactado: 24.5 bytes
Tempo gasto na compactacao: 0.30284 segundos
Taxa de compactacao: 51%
Tamanho do arquivo descompactado: 50 bytes
Tempo gasto para descompatar: 0.041085 segundos

----------/(arquivo medio)/----------

Testando uma entrada media de 500 bytes
Tamanho do arquivo compactado: 262.375 bytes
Tempo gasto na compactacao: 4.0626 segundos
Taxa de compactacao: 47.525%
Tamanho do arquivo descompactado: 500 bytes
Tempo gasto para descompatar: 0.180815 segundos

----------/(arquivo intermediario)/----------

Testando uma entrada media de 2000 bytes
Tamanho do arquivo compactado: 1065.125 bytes
Tempo gasto na compactacao: 13.509 segundos
Taxa de compactacao: 46.744%
Tamanho do arquivo descompactado: 2000 bytes
Tempo gasto para descompatar: 0.688649 segundos

----------/(arquivo grande)/----------

Testando uma entrada grande de 5000 bytes
Tamanho do arquivo compactado: 2665.875 bytes
Tempo gasto na compactacao: 35.786 segundos
Taxa de compactacao: 46.682%
Tamanho do arquivo descompactado: 5000 bytes
Tempo gasto para descompatar: 1.60374 segundos
```

Figura 6 - Resumo textual dos experimentos

[2] Ferramenta disponível pelo endereço: http://www.lipsum.com/

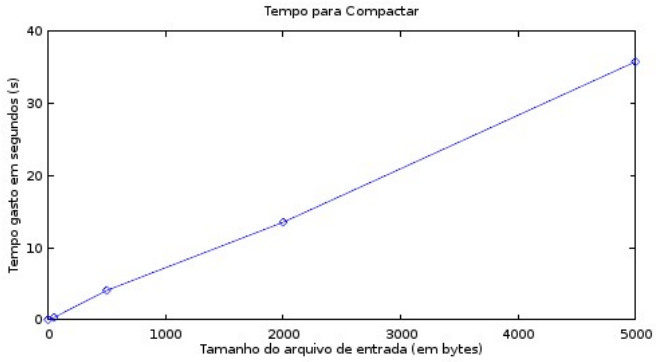

Figura 7 - Tempo gasto na compactação

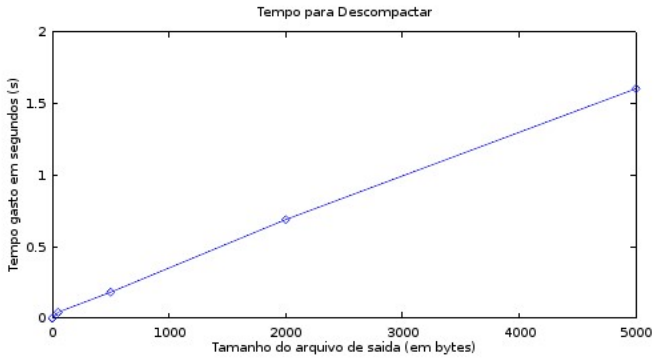

Figura 8 - Tempo gasto na descompactação

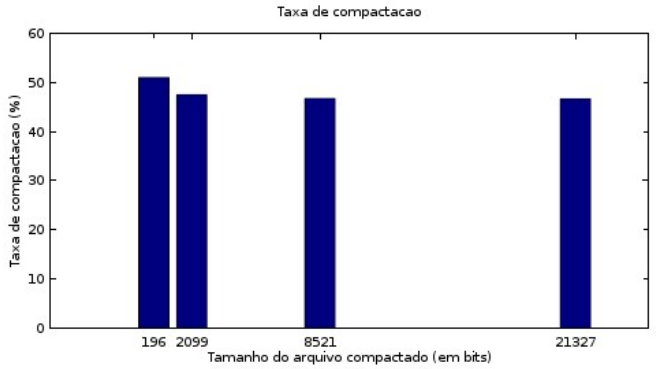

Figura 9 - Taxa de compactação

Após realizado os experimentos, os arquivos utilizados para compactação foram comparados com seus respectivos arquivos descompactados utilizando a ferramenta Meld[3]. Os arquivos estavam iguais, apresentando diferença com relação ao caractere de final de linha. O arquivo gerado pelo Octave utiliza o padrão LF, ao passo que o arquivo original, feito pelo bloco de notas, apresenta padrão CR-LF. Abrindo o arquivo de saída no Bloco de Notas o mesmo não apresentava as quebras de linhas. Outros processadores de texto que suportam o padrão LF, como o WordPad, Atom[4], ou SublimeText2[5], exibiram o arquivo com as quebras de linhas corretamente.

[3] Ferramenta disponível pelo endereço: http://meldmerge.org/.
[4] Ferramenta disponível pelo endereço: https://atom.io/.
[5] Ferramenta disponível pelo endereço: https://sublimetext.com/2.

V. CONCLUSÕES

O trabalho foi bastante motivador, por se tratar de uma implementação em uma linguagem de pouco domínio pelo autor, onde pode-se explorar a criatividade na utilização das estruturas a API que a linguagem oferece. A facilidade propiciada em plotar os gráficos e colher os resultados foram pontos positivos em relação ao uso do Octave.

Com relação as taxas de compactação apresentadas, obtivemos algo em torno dos 48% na média nos experimentos realizados, o que ficou na faixa daquilo que é comentado por Cormen et. al, que cita economias na casa de 20-90% como típicas [1]. Logo podemos entender, através dos resultados favoráveis, o porquê da técnica de Huffman ser amplamente utilizada nas mais diversas áreas, seja na compactação de arquivos de texto simples ou para arquivos de games, pacotes para transmissão em rede, músicas, imagens, etc.

REFERENCIAS

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Algoritmos: Teoria e Prática. Rio de Janeiro: Elsevier, 2012. 3ª edição.

[2] Wikipedia. David A. Huffman. Disponível em: https://pt.wikipedia.org/wiki/David_A._Huffman. Acesso em: Setembro de 2016.

[3] Wikipedia. ASCII. Disponível em: https://pt.wikipedia.org/wiki/ASCII. Acesso em: Setembro de 2016.

[4] P. D. Costa. A Codificação de Huffman. Disponível em: http://www.inf.ufes.br/~pdcosta/ensino/2016-2-estruturas-de-dados/material/CodificacaoHuffman.pdf. Acesso em: Setembro de 2016.

[5] Wikipedia . Codificação de Huffman. Disponível em: https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_de_Huffman. Acesso em: Setembro de 2016.