1 - Introdução:

O Algoritmo de Huffman é um método eficiente para compactação de dados, especialmente no caso de linguagem natural. Sua eficiência é ainda mais evidente em ambientes de recuperação de informação, pois ele apresenta algumas vantagens com relação a outros métodos de compressão:

- Manutenção de dicionário: apesar de gerar um maior gasto de memória, o fato de este algoritmo criar e manter a estrutura com o dicionário das substituições é bastante útil na hora de indexar as informações a serem recuperadas.
- Flexibilidade: não é necessário descomprimir todo o arquivo para realizar uma pesquisa num arquivo comprimido com este algoritmo, da mesma forma, partes do arquivo podem ser recuperadas sem necessidade da decodificação completa.

Além disso, o algoritmo de HuffMan é bem eficiente, sendo que o melhor caso com ordem de complexidade $N_{log}N$.

O Algoritmo de HuffMan possui várias implementações, mas o seu funcionamento básico é bastante simples: o objetivo é substituir os caracteres, palavras ou símbolos, dependendo do critério adotado, por códigos binários menores que aqueles que realmente representam tais símbolos. Para tanto, uma estatística de frequência de cada símbolo é o primeiro passo para a compressão. Em seguida, os símbolos mais frequentes recebem códigos binários menores do que aqueles menos frequentes.

Para alcançar este objetivo, a base doalgoritmo é a montagem de uma árvore binária indexada pelas frequências dos símbolos a serem substituidos. Essa árvore é montada de baixo para cima, sendo que a cada iteração os dois nodos menos frequentes são agrupados como filhos de um novo nodo, indexado pela soma das frequências dos dois filhos. Ao final de todas as iterações, os símbolos menos frequentes são as folhas mais extremas da árvore, e os símbolos mais frequentes as folhas mais próximas a raiz. Uma vez montada a árvore, ela é caminhada e a cada visita a um nodo é emitido o código binário que representa o caminho feito na árvore para chegar àquele nodo. Cada movimento para a esquerda gera um 0, e cada movimento à direita gera um 1.

2 - Estruturas de Dados:

É sabido que para alcançar o melhor resultado para o algoritmo de HuffMan, o ideal é que seja criada uma árvore de pesquisa para armazenar os nodos temporários enquanto o algoritmo os condensa. Isso se dá ao fato de que este algoritmo precisa constantemente retirar e inserir nodos dessa árvore afim de combiná-los e inseri-los como filhos de um novo nodo.

Porém a implementação com árvore de pesquisa demanda um tempo de desenvolvimento maior, que não estava disponível, portanto, por questão única de agilidade de produção, escolhi implementar uma lista encadeada para armazenar os nodos durante o processamento do algoritmo. Isso implicou num aumento da complexidade de tempo, como poderá ser observado na sessão 4.

A árvore do algoritmo em si é uma árvore binária simples, sem balanceamento, que contém nodos genéricos, estes por sua vez responsáveis por encapsular os nodos dotipo NodoBinário, que armazenam as informações necessárias para o processamento do algoritmo. No início existem na verdade várias árvores contendo apenas o nodo raiz, formando uma floresta com todos os caracteres presentes. A medida que o algoritmo é executado, estes nodos são condensados dois-a-dois, montando uma árvore única final.

3 - Implementações:

FilaDePrioridade.java:

Minha escolha para armazenamento dos nodos durante a execução do algoritmo foi da implementação de uma lista duplamente encadeada. Não há muito o que comentar sobre tal implementação, pois é uma versão bastante simplória e já conhecida. Inserção e Remoção usam apenas pequenas técnicas para diminuir o tempo de pesquisa, nada sofisticado.

O método extraiMenor() também não possui mistérios. Basicamente, tudo o que ele faz é varrer a lista a procura de um nodo que contenha o menor valor possível, e retorna esse valor no final. A implementação usada também é clássica e simplória: primeiro assume-se que o primeiro nodo é o menor de todos, sem seguida, a cada iteração verifica-se se o próximo nodo é menor que o atual, e caso for, substitui-se um pelo outro na seleção. No fim da iteração, o menor nodo disponível se manterá selecionado e é retornado.

HufEncoder.java

A classe HufEncoder possui apenas um método interessante: geraArvoreDeHuffman.

Escolhi uma implementação recursiva para este método, o que facilita em muito o entendimento do código. Em resumo, ao ser chamado o método extrai os dois menores nodos de uma lista de NodoBinario's, soma suas frequências e cria um novo nodo Z, cuja frequência é a soma obtida, possuindo os dois nodos anteriores como filhos. Ao final ele re-insere na lista a nova pequena árvore contendo os três nodos. O processo é repetido recursivamente até que só reste um nodo na lista, este será uma árvore contendo todos os nodos e as informações de caracteres e frequências já ordenadas. Esse é o coração do algoritmo de HuffMan, a montagem da árvore indexada pelas frequências dos caracteres do arquivo.

Codificador.java

A aplicação Codificador, apesar de ser a principal, não é a mais importante. Tudo que ela faz é receber as entradas e aplicar nelas os métodos contidos nas classes anteriores. No fim, um pequeno método recursivo de caminhamento na árvore imprime os nodos da árvore com seus respectivos caracteres e códigos obtidos. Para obter os códigos, a cada caminhamento para a esquerda um 0 é concatenado à string parâmetro, e a cada caminhamento para a direita, um 1 é concatenado a essa string. Ao chegar em uma folha, é impresso o valor de seu caractere e a string montada no caminhamento até ali.

4 - Análise de Complexidade:

No melhor caso, a complexidade de tempo de uma implementação clássica do algoritmo de Huffman é da ordem $N_{log}N$. Afinal ele precisa de N-1 iterações para montar a árvore, e $_{log}N$ iterações a cada condensação de nodos, para buscar na árvore binária os dois nodos menores para condensar.

Meu caso porém tem uma complexidade de tempo superior devido à escolha da Lista Encadeada para armazenamento dos nodos durante o processamento. Sabe-se que a complexidade de pesquisa numa lista encadeada é de ordem N, portanto a complexidade final de tempo da minha implementação do algoritmo de Huffman é da ordem de N^2.

5 - Extra: Contador De Caracteres

Para facilitar o desenvolvimento do programa, ajudar no tratamento de erros e na fase de testes, desenvolvi uma classe adicional chamada ContadorDeCaracteres. A função primordial dessa classe é simplória: contar a frequência de caracteres num arquivo de texto e retornar na saída a porcentagem da presença de cada caractere no texto. A classe é capaz de contar qualquer tipo de caracter, bastando configurar as variáveis abaixo:

- ALFA_SIZE – determina o tamanho do 'alfabeto' utilizado. Como escolhi usar apenas letras, e ignorando a caixa, o meu ALFA_SIZE está configurado para 26 símbolos.
- TO_FIRST_CHAR – representa a distância do caractere escolhido para ser o primeiro do 'alfabeto' para o verdadeiro primeiro caractere da tabela ASCII.
- FIRST_CHAR – determina qual será o primeiro caractere do 'alfabeto'

O que essa classe faz basicamente é fornecer para o Codificador a informação do tamanho do 'alfabeto' utilizado, através da variável ALFA_SIZE, isso permite que o algoritmo final trabalhe com letras, números, caracteres especiais, ou até mesmo com todos os caracteres disponíveis na tabela ASCII, se assim for desejado.

O funcionamento também é bastante simples, entra-se com um arquivo de texto simples, tomando cuidado para que ele contenha apenas caracteres presentes no 'alfabeto' configurado. Primeiramente ele gera uma tabela com o número de vezes em que cada caractere aparece no arquivo, e em seguida gera uma nova tabela contendo a representação percentual destas frequências, essa tabela final é impressa em um arquivo de saída, que deve ser encaminhado para o programa principal Codificador, para cálculo da árvore de Huffman correspondente.

6 – Anexos:

Estão incluidos dois anexos neste documento:

1. Código: Contém o código fonte de todas as classes utilizadas
2. Testes: Contém as entradas para o programa ContadorDeCaracteres, as saídas geradas por ele, e os resultados gerados pelo Codificador a partir dessas saídas.