Trabalho Prático 3 – Comprimir e Descomprimir Arquivo

Gustavo Dias Apolinário - 2022035911

Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brasil

gustavo-apolinario@ufmg.br

1. Introdução

O problema proposto foi a implementação de um programa que fosse capaz de compactar os arquivos de texto de um jornal através do **algoritmo de Huffman**. Obviamente, é necessário descompactar para acessar a informação posteriormente, portanto procuramos a dupla funcionalidade do programa. É relevante ter em mente que para esse programa o algoritmo compacta em ordem de letra.

O programa foi desenvolvido na linguagem C++, a partir do compilador GCC da GNU Compiler Collection. Sistema Operacional: Linux Ubuntu 22.04.01 LTS.

2. Método

O programa utilizou as seguintes estruturas de dados: Árvore Huffman, Fila de Menor Prioridade, Árvore Mapa, Nó Huffman e Nó Mapa.

É importante ressaltar os motivos que levaram à escolha desss estruturas, principalmente, Árvore de Huffman, Fila de Menor Prioridade e Árvore Mapa. Para isso, é necessário compreender a ideia do algoritmo de Huffman;

Algoritmo de Huffman:

O algoritmo de Huffman é um método de compressão de dados que utiliza uma abordagem baseada na frequência de ocorrência dos caracteres em um determinado conjunto de dados. A ideia central do algoritmo é atribuir códigos de tamanho variável aos caracteres de entrada, de modo que os caracteres mais frequentes sejam representados por códigos menores, enquanto os caracteres menos frequentes sejam representados por códigos maiores. A compactação ocorre a partir dos binários gerados, uma quantidade menor de caracteres é gerado a partir dos novos binários, de forma que no documento original existe um texto, esse texto é transformado em 0´s e 1´s, esses 0´s e 1´s formam um novo texto sem sentido, mas com tamanho reduzido e a partir desse texto sem

sentindo se recupera o texto original. Exemplo na observação no Tópico Análise de Complexidade.

Árvore Huffman:

Em conjunto com o Nó Huffman, essa árvore é utilizada para gerar esses códigos de tamanho variável de forma eficiente. A árvore de Huffman é uma árvore binária completa, onde cada nó interno representa a soma das frequências dos caracteres em seus nós filhos. Os caracteres são armazenados nas folhas da árvore, de maneira que além de gerar a codificação também armazena essa informação de forma eficiente.

Fila de Menor Prioridade:

Durante a construção da árvore de Huffman, os caracteres são agrupados em pares de menor frequência, e um novo nó é criado para representar a soma das frequências desses caracteres. Esse processo é repetido até que todos os caracteres sejam agrupados em um único nó raiz.

Essa dinâmica de agrupar pares de menor frequência implica duas características: é preciso sempre manter ordenado do menor para o maior a fim de conseguir gerar o próximo nó e é preciso somar os dois nós menos frequentes e colocar esse resultado de volta na ordem. Logo, a Fila de Menor Prioridade, desempenha a função de remover os dois nós menos frequentes, pois estes estarão sempre na frente pela propriedade de Menor Prioridade que mantém a ordem dentro dela.

Árvore Mapa:

A Árvore Mapa é utilizada para armazenar o código. Primeiramente, é necessário ter em mente a partir da Árvore de Huffman que se extrai o código, entretanto ela não armazena esse valor. É evidente que é uma possibilidade extrair o código a partir dela toda vez que precisarmos dessa informação, contudo como além dos caracteres nas folhas existem os nós que representam somas de frequências é possível que para um volume grande de caracteres únicos esse caminhamento seja pouco eficiente devido ao grande número de nós. Nessa perspectiva é benéfico armazenar esses códigos em outra estrutura de dados com uma pesquisa eficiente e que a partir de uma chave como a letra possamos encontrar o código correspondente.

Essa necessidade é suprida perfeitamente pela Árvore Mapa, a qual possui as seguintes propriedades: é uma árvore binária de pesquisa, é possível fazer buscas através da chave para encontrar o valor correspondente, é pseudo balanceada por meio do algoritmo de AVL. Resumidamente, ela é uma estrutura que mantém as informações bem distribuídas, o que facilita a busca, além disso é possível armazenar pares em conjunto de forma que ao buscar um valor chave obtêm-se o seu par.

Dessa forma, para a compactação essa estrutura de dados é utilizada para armazenar e “traduzir” de caractere para codificação em binário com maior eficiência. Entretanto, não é útil para descompactação, pois nessa fase utiliza-se o método menos eficiente de percorrer a árvore, uma vez que a estratégia do mapa não funciona para arquivos sem marcações, como o caso do binário. Não é possível procurar uma chave se não temos ela marcada, logo se realiza o caminhamento até encontrar um caractere e depois esse processo recomeça até finalizar a sequência de bits.

3. Análise de Complexidade:

Complexidade de Tempo:

Algoritmo de Huffman: **O (n*log n).** A construção da árvore de Huffman envolve a seleção e fusão repetida de nós com as frequências mais baixas. Essa etapa foi implementada através Fila de Menor Prioridade, resultando em uma complexidade de tempo de O (n * log n), onde n é o número de símbolos únicos na entrada. Esse é o processo mais custoso do algoritmo. A Fila de Menor Prioridade gera essa complexidade, pois cada inserção e remoção é O (log n), uma vez que os elementos são armazenados em um heap binário, que é uma árvore binária completa com a propriedade do heap. Sendo que fazemos esse processo a cada novo caractere, dessa forma resulta em O (n * log n), sendo n cada caractere único.

Complexidade de funções importantes:

Observação: Entenda como Bitstring, uma string que só armazena 0´s e 1´s como caracteres e o binário como a representação desses 0´s e 1´s a cada 8 bits. Exemplo: Bitstring: “0010110”, binário correspondente: ,(vírgula). Esse é o principal processo para compactar os arquivos! “00” pode representar ‘a’, “101” ‘b’ e “10” ‘c’. Dessa forma, 3 caracteres passam a ser apenas um!

Função de Codificação: O (n). É necessário gerar o código para cada caractere único e inseri-lo na Árvore Mapa, como o processo deve percorrer todos os caracteres é O (n).

Função de Leitura: O (n). Ao descompactar o programa lê cada número para identificar o código e encontrar o caractere correspondente, esse processo lê número por número, todos os 0´s,1´s, portanto O (n). Sendo n, todos os caracteres (números) do código, repare que se a compressão for boa esse n pode ser muito pequeno, o mesmo vale para o contrário.

Pesquisa Compactação: O (log n). Devido a implementação da Árvore Mapa e todas as propriedades já explicadas ela possui o desempenho de pesquisa e remoção de uma Árvore AVL, portanto O (log n), n sendo o número de elementos na árvore.

Formação de Bitstring Compactação: O (n * log k). Nessa etapa, para cada caractere, iremos buscar seu valor em código correspondente na Árvore

Mapa e acrescentá-lo na Bitstring. Portanto, faremos buscas com custo O (log k), sendo k o número de elementos na árvore e n quantos caracteres.

Conversão Binária: O(n). Sendo n, o tamanho da bitstring formada ou do binário formado, pois tanto na compactação quanto na descompactação é necessário percorrer elemento a elemento ou da bitstring ou do resultado binário para gerar o resultado.

Função de Escrita: O(n). Sendo n o tamanho da bitstring, segue a mesma lógica que a Leitura, irá percorrer cada bit para encontrar o caractere correspondente e gerar o texto. Lembre-se que na escrita da descompactação não ocorre a pesquisa na árvore e sim refazendo o caminho para cada bit.

Complexidade de Espaço:

Algoritmo de Huffman: O (n). É utilizado um espaço adicional para armazenar cada caractere único, tanto na Fila, quanto nas Árvores, portanto O (n) apenas.

Função de Codificação: O (n). Nessa função se insere na Árvore Mapa os caracteres com seus códigos correspondentes, portanto O (n).

Função de Leitura: O (1). Essa função apenas utiliza variáveis de tamanho fixo para armazenar valores temporários para facilitar a leitura e pesquisa.

Pesquisa Compactação: O (1). A função de pesquisa em si não gasta nenhum espaço, entretanto a estrutura árvore gasta O(n) como citado anteriormente.

Formação de Bitstring Compactação: O (n). Uma string armazena a Bitstring e o tamanho que ela ocupa varia de acordo com a quantidade de caracteres do arquivo texto de entrada, sendo essa quantidade n.

Conversão Binária Compactar: O(1). Como para compactar escreve-se diretamente no arquivo, não há armazenamento nessa função.

Conversão Binária Descompactar: O(n). Para cada bit n lido, é preciso armazená-lo em uma string para manipulá-lo nas próximas etapas

Função de Escrita: O(1). Por escrever diretamente no arquivo, não há armazenamento.

4. Estratégias de Robustez:

Como estratégias de programação defensiva algumas medidas foram tomadas:

- Conferir se o arquivo foi aberto devidamente e gera erro se necessário.
- Impossibilitar acesso à endereços nulos.
- Todo ponteiro é colocado como nulo na criação.
- Avisa se o valor procurado não existir na Árvore Mapa, não gera um erro, mas avisa. Interromper o programa por conta disso seria um exagero, pois em uma situação real poderiam haver erros e fugiria do contrato de um mapa.

- Confere se o valor do código é válido para conversão sob a mesma premissa do item anterior, não emite um erro.
- Confere se o arquivo está no formato adequado (pós compactação), se necessário gera erro.
- Confere se a Fila está vazia na tentativa de remoção, se necessário gera erro.
- Caso não forem passados os arquivos gera erro.

5. Análise Experimental:

Taxa de Compressão:

É interessante analisar que a eficiência do algoritmo de Huffman está diretamente ligada à distribuição de caracteres, uma vez que os caracteres utilizados mais frequentemente apresentam uma representação binária menor por estarem mais próximo da raiz da Árvore de Huffman. Consequentemente, se todos os caracteres possuírem a mesma frequência, alguns deles terão uma representação binária extensa que irá se repetir muitas vezes, diminuindo a taxa de compressão. Para comprovar esse fenômeno, foram realizados os seguintes teste:

Arquivo 1: Todos os caracteres ASCII repetidos, igualmente, inúmeras vezes. A expectativa é que a taxa de compressão seja baixa, além disso, demonstra a eficiência do algoritmo para qualquer caractere. Resultado: 15,14% de compressão.

Arquivo 2: Um arquivo muito grande e aleatório. Por ser aleatório e com uma grande amostra, tende a ter a mesma frequência para todos os caracteres, além disso, demonstra a capacidade do algoritmo de lidar com grandes entradas. Resultado: 16,72% de compressão.

Arquivo 3: Um arquivo médio apenas com um caractere, a letra ‘a’. Nesse caso era esperado uma boa taxa de compressão, pois toda a frequência está concentrada em um único caractere. Resultado: 87,07% de compressão.

Arquivo 4: Texto de 100.000 bytes de Lorem Ipsum, um texto em latim utilizado em design gráfico com o intuito de manter a neutralidade e não dar qualquer viés. Um equilíbrio entre o aleatório e o provável, sendo adequado para representar um caso médio de texto. Resultado: 45,92% de compressão.

Portanto, é seguro afirmar que o conteúdo do texto, principalmente, a distribuição da frequência das palavras afeta fortemente a qualidade da compressão. Ademais, o programa se comporta como esperado, apresentando resultados bons para seu propósito.

Tempo:

Arquivo 1:

Tamanho (Entrada/Binário): 185/155

Tempo total Compactação: 40.3401 milissegundos

Tempo Huffman: 0.0657 milissegundos

Tempo Formação Bitstring: 21.9767 milissegundos

Tempo de Conversão para Binário e escrita: 8.317 milissegundos

Tempo total Descompactação: 44.918 milissegundos

Tempo Huffman: 0.0308 milissegundos

Tempo de Leitura: 0.0012 milissegundos

Tempo para Desconverter o Binário: 21.6348 milissegundos

Tempo de Escrita: 16.8607 milissegundos

Arquivo 2:

Tamanho (Entrada/Binário): 724/603

Tempo total Compactação: 134.542 milissegundos

Tempo Huffman: 0.0716 milissegundos

Tempo Formação Bitstring: 86.2693 milissegundos

Tempo de Conversão para Binário e escrita: 30.0202 milissegundos

Tempo total Descompactação: 163.489 milissegundos

Tempo Huffman: 0.0258 milissegundos

Tempo de Leitura: 8.3547 milissegundos

Tempo para Desconverter o Binário: 88.2558 milissegundos

Tempo de Escrita: 64.1101 milissegundos

Arquivo 3:

Tamanho (Entrada/Binário): 116/15

Tempo total Compactação: 11.7418 milissegundos

Tempo Huffman: 0.0048 milissegundos

Tempo Formação Bitstring: 5.8514 milissegundos

Tempo de Conversão para Binário e escrita: 0.6623 milissegundos

Tempo total Descompactação: 8.8954 milissegundos

Tempo Huffman: 0.0004 milissegundos

Tempo de Leitura: 0.0003 milissegundos

Tempo para Desconverter o Binário: 1.8999 milissegundos

Tempo de Escrita: 3.5843 milissegundos

Arquivo 4:

Tamanho (Entrada/Binário): 98/53

Tempo total Compactação: 18.2962 milissegundos

Tempo Huffman: 0.0325 milissegundos

Tempo Formação Bitstring: 9.652 milissegundos

Tempo de Conversão para Binário e escrita: 2.7624 milissegundos

Tempo total Descompactação: 18.2562 milissegundos

Tempo Huffman: 0.0108 milissegundos

Tempo de Leitura: 1.2522 milissegundos

Tempo para Desconverter o Binário: 7.7454 milissegundos

Tempo de Escrita: 6.4834 milissegundos

É possível comprovar algumas análises feitas anteriormente com esses dados.

Note que o Huffman em 1 e 2 são muito maiores que em 3 e 4, pois eles cobrem todo o ASCII, logo possuem um número maior de caracteres únicos.

A Formação de Bitstring, em 1 e 2 são muito diferentes, pois apesar de terem o mesmo log k da Árvore Mapa por terem a mesma quantidade de caracteres únicos, o n se distingue muito para cada arquivo devido ao tamanho superior do segundo. Além disso, a Formação de Bitstring no arquivo 3 foi muito mais rápida devido ao log k ser mínimo.

Por consequência, o tamanho pequeno da Bitstring e uma boa compressão que gerou um binário pequeno do arquivo 3, fez com que ele tivesse um tempo para converter e desconverter para binário muito pequeno.

Por fim, o tempo de leitura e escrita, foram proporcionais às Bitstrings como esperado.

6. Conclusões:

Foi possível observar as vantagens e desvantagens do algoritmo de Huffman. O programa se tornou eficiente para casos em que existem poucos caracteres que se repetem muitas vezes, de forma que a frequência esteja concentrada. Em textos médios o desempenho é razoável, principalmente, pelo fato de que vogais serem amplamente repetidas, em português. Por ter um complexidade O(n * log n) para criação da Árvore e O(n * log k) para formação de Bitstring não é possível afirmar que o desempenho dele seja muito eficiente. Ademais, a taxa de compressão pode ser muito ruim nos piores casos. Portanto, a conclusão é que ele é um bom algoritmo apenas para casos específicos.

7. Referências Bibliográficas:

CORMEN, Thomas H.. **Introduction to Algorithms**. 4. ed. Massachusetts: The MIT Press, 1989. ISBN 9780070131514.

BARNWAL, Aashish . Geeks for Geeks. **Huffman Coding | Greedy Algo-3,** 2021. Disponível em: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/. Acesso em: 28 jun. 2023.

INTERNO 15. Lore Ipsum. Lore Ipsum, 2018. Disponível em: https://www.loremipzum.com/pt/gerador-de-texto. Acesso em: 01 jul. 2023.

STANS, Jelly. Generate Random ASCII. **Caton Mat,** 2018. Disponível em: https://catonmat.net/tools/generate-random-ascii. Acesso em: 01 jul. 2023.

8. Instruções para Compilação e Execução:

- Abrir o terminal
- Acessar a pasta TP
  Comando: cd / <caminho para a pasta /TP >
- Compilar com o Makefile Comando: make
- Executar com o Makefile Comando: make run
- Finalizado, caso deseja limpar: Comando: make clean

- Importante ressaltar que os arquivos de texto a serem lidos devem estar na raiz do projeto e quando acrescentados devem ser adicionados aos makefile para limpar e executar.

  Para limpar, na seção clean, adicione:

rm <nome_arquivo_binario> <nome_arquivo_saida>

Para executar, na seção run, adicione:

$(EXECUTABLE) -c <nome_arquivo_entrada> <nome_arquivo_binario>

$(EXECUTABLE) -d <nome_arquivo_binario> <nome_arquivo_saida>