Universidade Federal Do Espírito Santo, Centro Tecnológico

Maria Luiza Ferreira Reis Santos

Raony Togneri Gomes

Huffman: análise da compactação e descompactação de arquivos

Vitória, 2024

Sumário

1. Introdução

Atualmente, cerca de 66% da população mundial possui acesso a internet, tornando meios de comunicação uma das principais ferramentas na sociedade, tanto para o uso profissional, quanto o uso pessoal. Entretanto, os canais de comunicação enfrentam um problema comum: a quantidade de arquivos a tratar. Uma solução possível é a compressão de arquivos, reduzindo o tamanho necessário para armazenamento e, podendo ser descomprimido posteriormente, sem qualquer perda de informação.

Dito isso, este trabalho tem como objetivo a compactação e descompactação de arquivos, como textos e imagens, seguindo o que foi ensinado na disciplina de Estrutura de Dados.

2. Desenvolvimento:

Para o desenvolvimento, seguimos o passo a passo para gerar a codificação Huffman: gerar um vetor ordenado com os pesos e as árvores a partir destes, somando os pesos até chegarem todos em um nó árvore comum.

Com isso, foi atribuído a cada caractere utilizado no arquivo uma sequência de bits - os caracteres com menor frequência possuem maiores sequências de bits, já os com maior frequência, possuem menores sequências de bits - e gerado o arquivo compactado com a árvore binária e o arquivo compactado.

Assim, é possível recuperar a árvore binária afim de percorrer a sequência de bits do arquivo e, simultaneamente, a árvore para decodificação do arquivo.

2.1 Tipos de dados estruturados:

Para implementação do programa, implementamos alguns tipos de dados e utilizamos o tad bitmap disponibilizado. A seguir, os arquivos utilizados: tree.c, bitmap.c, compacta.c, descompacta.c.

Mesmo no tad Bitmap, que foi cedido pela professora, implementamos alterações nele para que realizasse mais funções necessárias para o funcionamento do programa.

2.2 Funcionamento geral do programa:

Para execução do programa, recomenda-se seguir os seguintes passos no terminal:

```
Make compc
./compacta <arquivo a ser compactado>

Make compd
./descompacta <arquivo gerado pelo compactador>
```

O comando *Make compc* compila o compactador.c e suas dependências. De forma análoga, o *Make compd* compila o descompactador.c e suas dependências.

Exemplificando: caso queira compactar uma imagem e o nome do arquivo da imagem seja *exemplo.png*, o usuário deverá digitar conforme abaixo:

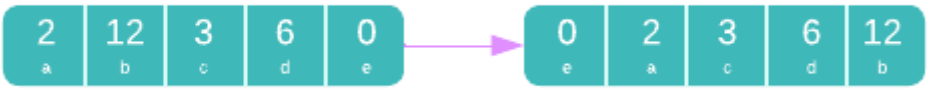
```
Make compc
./compacta exemplo.png

Make compd
./descompacta exemplo.png.comp
```

Nota: o arquivo passado para descompactar terá sempre o nome do arquivo original acrescentado com *.comp*. Neste caso, *exemplo.png.comp*.

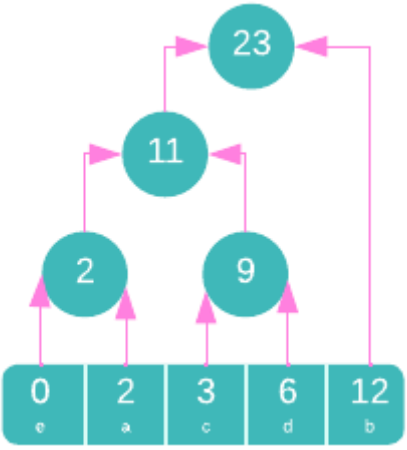
O arquivo de saída será gerado com o sufixo *desc_***.** No exemplo, a saída será *desc_exemplo.png*.

2.3 Compactação:

Para compactação, foi criado e ordenado um vetor de inteiro para armazenar o peso de cada caractere ascii presente no arquivo a compactar. Além disso, foi adicionado também um caractere extra para representar a parada do arquivo.

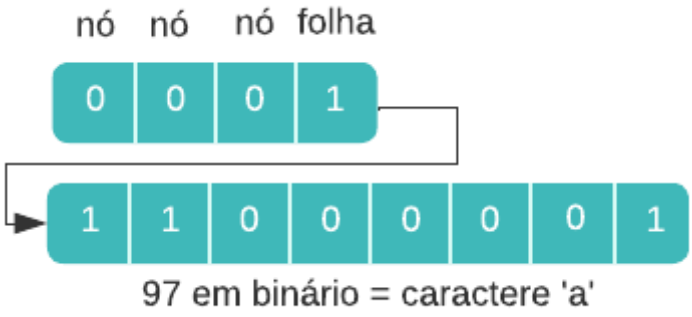


Com o vetor ordenado, foi criado um nó de árvore binária para cada índice, onde os pesos serão somados e ordenados (em ordem crescente) pela função *OrganizaArvorePorPesos* para criar outro nó até que haja apenas um nó pai comum de todos os índices anterior vetor.



Agora, com a árvore completa, criamos um vetor de mapa de bits, onde as sequências de bits para chegar em cada caractere foram armazenadas no mesmo índice em onde elas pertenciam no vetor ordenada anterior, para reduzir o tempo de execução do programa. Para concluir, podemos chamar a função Compacta: ela irá ler novamente o arquivo a ser compactado e gerar um mapa de bits de acordo com a codificação de cada caractere.

Além disso, ela também irá gerar o mapa de bits da árvore com a função *PreencheBitmapArvore* para reduzir o tamanho necessário para salvar ela



.

Em seguida, será gerado um arquivo binário que será salvo no caminho passado por parâmetro na execução do programa, onde serão escritas as seguintes informações, com a função *ImprimeBitmapArquivo*, nesta exata ordem: tamanho do mapa de bits da árvore, o mapa de bits da árvore, a quantidade de caracteres de parada presentes no arquivo, o mapa de bits do arquivo.

O motivo da impressão da quantidade de caracteres é porque, em arquivos que não são texto, ocorreram problemas com o nosso caractere de parada: ele se repetia mais de uma vez, fazendo a descompactação apenas até sua primeira ocorrência. A inserção manual do caractere ao final do arquivo garante que, no n-ésimo caractere de parada, será definitivamente o final do arquivo.

2.4 Descompactação:

Para descompactar, recebemos o caminho do arquivo binário por parâmetro na execução do programa e abrimos para recuperar a árvore binária com a função *RecuperaArvore*, onde ela irá ler o tamanho do mapa e o mapa, além de preencher a árvore binária com a função *ColocandoConteudoArvore*, que percorre o mapa e se desloca nela da esquerda à direita até encontrar um nó folha (um nó sem filhos).

Com a árvore recuperada, geramos o arquivo de saída, lemos a quantidade de caracteres de parada presentes no mapa de bits do arquivo e decodificamos o texto com a função *DecodificaTexto*. Essa função lê o mapa de bits de byte a byte com a função *LerTextoBinArquivo*, até que encontre os N números de caracteres de parada, retornando cada caractere com a função *BuscaLetraEmArvore*, que percorre a árvore de acordo com a sequência de bits passada, e preenche o arquivo de saída com o caractere.

Por fim, temos o arquivo decodificado e descompactado, podendo então chamar as funções de liberação de memória e encerrar o programa.

3. Conclusão:

Todas as requisições do trabalho foram atendidas, de forma que tanto o tempo de execução e a taxa de compressão foram satisfatórias.

Durante o trabalho, um dos maiores desafios foi trabalhar com bits, devido a limitação em C e a falta de experiencias anteriores e otimizar o uso de memória para a compactação. Houveram também alguns problemas com a verificação de memória devido liberações duplas ocorrendo em casos de lotação no mapa de bits.

Entretanto, foi um trabalho par, somou experiência à ambos com o trabalho em equipe e o uso de ferramentas para trabalhar com bits, além de aplicar parte do conteúdo aprendido em sala de aula.

4. Referências

QuickSort on Singly Linked List. Disponível em: <https://www.geeksforgeeks.org/quicksort-on-singly-linked-list/>.

PREPBYTES. QuickSort on Singly Linked List. Disponível em: <https://www.prepbytes.com/blog/linked-list/quicksort-on-singly-linked-list/>.