Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estruturas de Dados 2019/2
Prof. Carlos Teixeira, Luiz Chaimowicz, Olga Goussevskaia
Monitores: Álisson R. Svaigen e Matheus A. Diniz

Trabalho Prático 3: O problema de compressão de mensagens de Rick Sanchez

Valor: 10 pontos
Data de entrega: 27 de novembro de 2019

Introdução

Com sua agenda de viagens organizada, Rick foi capaz de conhecer várias pessoas (e alienígenas) de diferentes planetas e galáxias. Mensagens de texto são o principal meio de comunicação entre Rick e seus novos amigos devido à enorme distância física entre eles. Evidentemente, planos de dados intergaláticos são muito caros e portanto, Rick deseja diminuir o tamanho de suas mensagens para que não precise comprar pacotes de dados para continuar conversando com seus amigos. Para isso, ele pediu sua ajuda para criar um algoritmo capaz de comprimir suas mensagens de texto, e assim, diminuir a quantidade de dados enviados.

Detalhes do problema

Neste trabalho, serão praticados conceitos relacionados a estruturas eficientes de busca. Para isso, será pedida a criação de um programa capaz de analisar mensagens de texto e produzir uma codificação eficiente do texto, reduzindo o número de `bits` necessários para codificar a mensagem.

A principal intuição por trás da compressão é utilizar códigos curtos para palavras que são muito utilizadas na mensagem, e códigos mais longos para palavras pouco utilizadas. A codificação ASCII, padrão da linguagem C, não segue essa intuição pois cada caracter é codificado por 8 bits, e portanto, o tamanho da codificação de uma palavra depende apenas da quantidade de caracteres. Assim, a codificação ASCII não é ótima no quesito compressão, visto que palavras longas (ex.: computador) podem ser frequentes no texto e ainda sim possuirem uma codificação muito longa.

Portanto, num primeiro momento, seu programa deverá analisar a mensagem para descobrir quais palavras são mais frequentes naquele texto. Em seguida, seu programa deverá criar a nova codificação das palavras através de uma estrutura de árvore que irá priorizar as palavras mais frequentes com códigos mais curtos. Mais detalhes sobre essa estrutura na seção de **Exemplos**.

Entrada e Saída

A entrada será composta por várias linhas e deverá ser lida pela entrada padrão do sistema (`stdin`). A primeira linha irá conter um inteiro **N** ($1 < \mathbf{N} < 2^{31}$) indicando o número de palavras contidas na mensagem de texto. A segunda linha irá conter **N** palavras (que podem conter até 63 caracteres, cada) separadas por espaço, representando a mensagem de texto. Além do espaço, somente os caracteres de `a`–`z` estarão presentes na mensagem, ou seja, a entrada não irá conter letras maiúsculas, sinais de pontuação ou caracteres acentuados.

As próximas linhas serão compostas por um caractere **I** e uma palavra **P** (**P** aparece na mensagem). Quando o caractere **I** assumir o valor '`q`', o programa deverá produzir uma saída indicando quantas vezes a palavra **P** aparece na mensagem. Já para o valor '`c`', o programa deverá produzir uma saída com a codificação comprimida da palavra **P**. Esses resultados deverão ser impressos na saída padrão do sistema (`stdout`). O final da entrada é indicado por `EOF` (`CTRL + D` em um terminal Unix).

Exemplos

Entrada
22 nao acho que quem ganhar ou quem perder nem quem ganhar nem perder vai ganhar ou perder vai todo mundo perder perder q perder c perder q ganhar q nao c nao
Saida
5 01 3 1 0010

Figura 1: Exemplo de entrada e saída

Medindo a frequência das palavras

Em um primeiro momento, seu programa deverá medir quantas vezes cada palavra aparece na mensagem. Para fazer tal medição, basta percorrer a mensagem e procurar cada palavra em uma estrutura de busca. Se a palavra não for encontrada, então esta é a primeira vez que ela apareceu na mensagem, e portanto, deverá ser inserida na estrutura e associada a um contador inicializado com valor 1. Caso a palavra já exista na estrutura, basta incrementar o contador associado a essa palavra. Quando toda a mensagem já estiver processada, esse contador irá refletir quantas vezes cada palavra apareceu na mensagem.

A estrutura de busca implementada deverá ser eficiente, visto que a mensagem pode conter muitas palavras e que ao menos uma busca será realizada para cada palavra. Sugestões de estruturas de busca eficientes incluem:

- tabelas hash (eficiente e implementação fácil)
- TRIE (eficiente em tempo mas ocupa muita memória, implementação moderada)
- árvore binária (eficiência moderada, implementação moderada)

Além do contador, cada nó também pode armazenar um atributo string para a nova codificação da palavra que será criada na próxima etapa. A Figura 2 mostra um exemplo de estrutura de busca com tabela hash usando listas encadeadas para tratar as colisões.

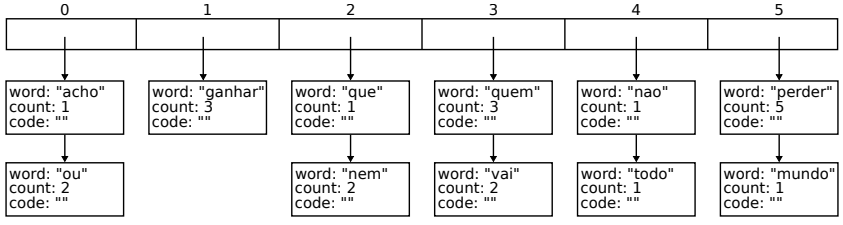

Figura 2: Tabela hash para a entrada da Figura 1. Nesse exemplo, a tabela possui tamanho 6 para facilitar a visualização, mas tabelas maiores podem diminuir o número de colisões.

Criando uma nova codificação para a mensagem

Para reduzir o tamanho da mensagem como um todo, palavras muito utilizadas devem possuir códigos curtos enquanto palavras pouco utilizadas podem possuir códigos mais longos. Especificamente, o algoritmo desenvolvido deverá implementar a codificação de Huffman. Nessa codificação, cada palavra será representada em uma folha de uma árvore binária. O código para cada palavra é então obtido através do caminhamento da árvore. Para cada aresta tomada à esquerda, um bit `0` é adicionado à codificação, e para cada a aresta à direita, um bit `1`. A Figura 3 mostra a codificação de Huffman em formato de árvore binária para a mensagem de exemplo anterior. O caminhamento para a palavra 'perder' produz o código `01`, já o caminhamento para a palavra 'mundo', o código `0001`.

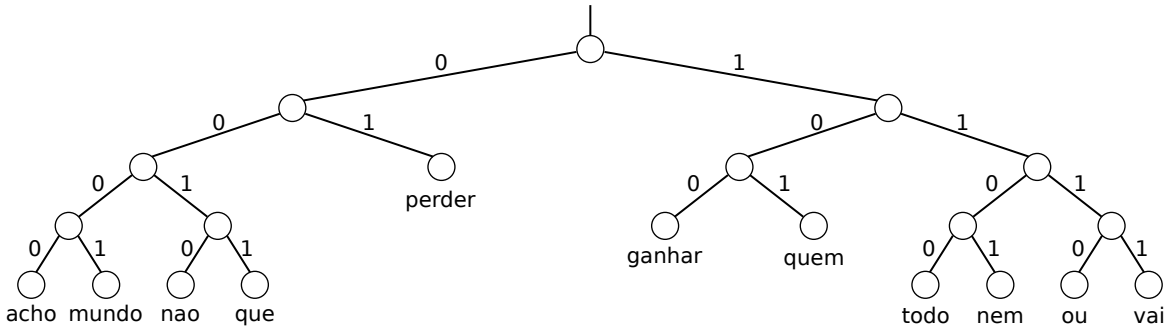

Figura 3: Árvore binária obtida pelo algoritmo de Huffman para a mensagem de exemplo da Figura 1

A construção dessa árvore é feita de maneira bottom-up: inicialmente, cada palavra é representada por uma árvore de um elemento, e então as árvores são combinadas até que uma única árvore contenha todas as palavras. A ordem em que as árvores são combinadas determina o formato final da árvore, e portanto, o tamanho do código de cada palavra. Para que os códigos mais longos fiquem associados a palavras pouco frequentes, a cada iteração, as duas árvores que possuem os menores contadores devem ser combinadas. Veja que esse processo de construção pede por uma estrutura de busca que determine qual o menor elemento a cada iteração.

As Figuras 4, 5 e 6 mostram todas as iterações necessárias para a construção da árvore para a mensagem de exemplo[1]. Inicialmente, começamos com uma floresta onde cada árvore contém um único nó representando uma das palavras da mensagem. A cada iteração, as duas árvores com menor frequência são combinadas de forma que a menor delas seja conectada à esquerda da raiz na nova árvore. Essa nova árvore é então re-inserida na floresta, sendo sua frequência a soma das frequências das árvores combinadas.

Em determinadas iterações, várias árvores na floresta podem possuir a mesma frequência, existindo, portanto, várias combinações válidas. Nesse casos, deve-se utilizar o número de folhas como fator de desambiguação. Isto é, caso duas árvores possuam a mesma frequência, aquela com menor número de folhas será a menor delas. Caso a ambiguidade se mantenha (ex.: Iteração 1) ela deverá ser resolvida pela ordem lexicográfica da menor palavra contida na árvore. Isto é, dadas duas árvores com mesma frequência e número de folhas, aquela que possuir, em uma de suas folhas, a palavra de menor ordem lexicográfica será a menor delas. Ao combinar duas árvores, pode ser interessante armazenar nos nós intermediários a menor palavra e o número de folhas da árvore para facilitar a computação das desambiguações.

Com a árvore já finalizada, pode-se perceber que a obtenção do código de uma palavra específica é de difícil obtenção, pois não é possível saber qual caminho deve ser tomado para alcançar a folha que corresponde àquela palavra. Para otimizar a busca dos códigos, basta fazer um único caminhamento na árvore (mantendo a informação de quais arestas foram tomadas) e, a cada folha visitada, armazenar o código obtido na estrutura da etapa anterior, como a tabela da Figura 2.

[1]Para simplificar a visualização nas figuras, utilizamos um vetor ordenado para determinar o menor elemento a cada iteração. Porém, outras estruturas, como a `Heap`, podem ser mais adequadas e eficientes.

Figura 4: Iterações necessárias para a construção da árvore da mensagem de exemplo (1/3).

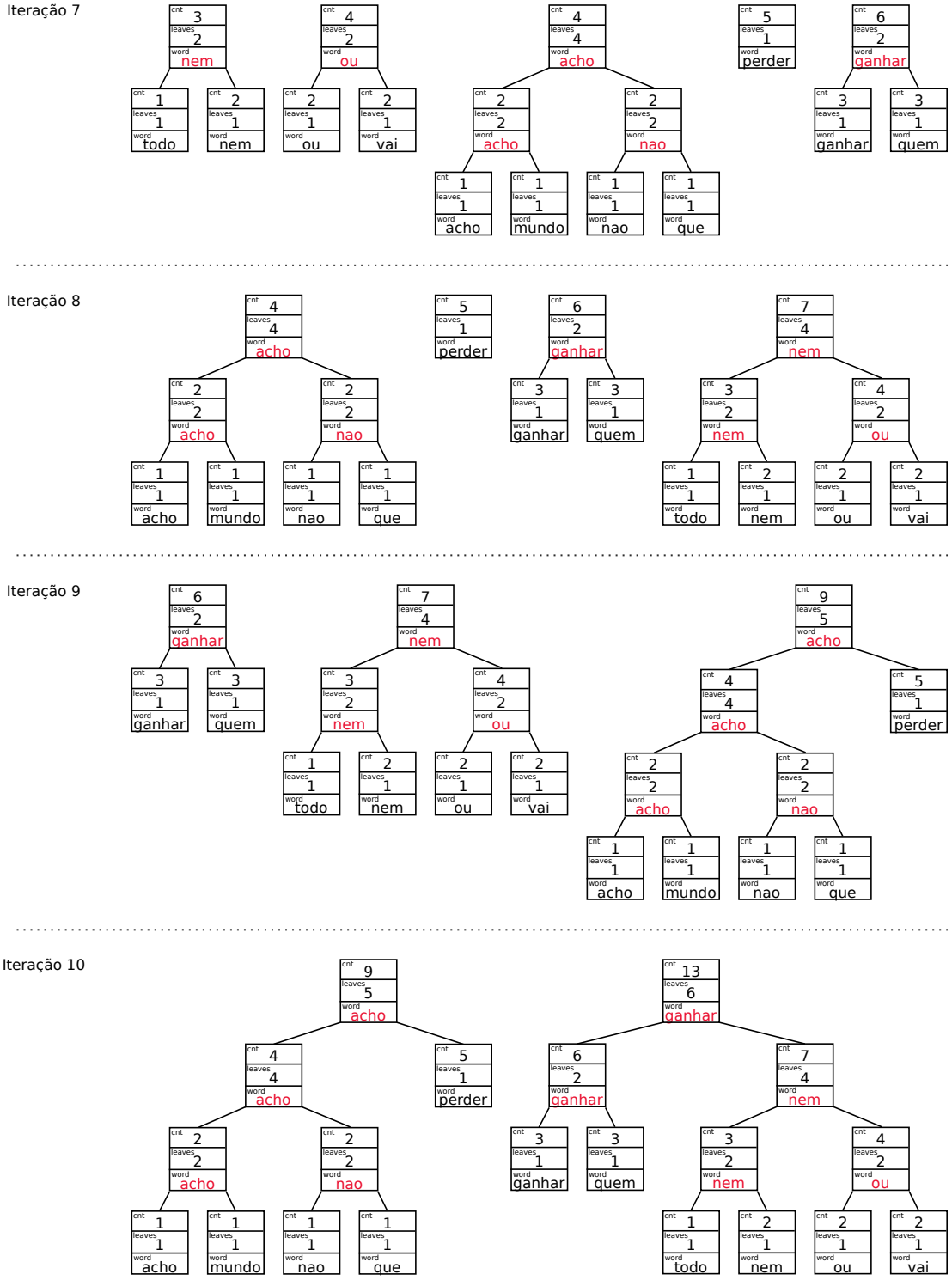

Figura 5: Iterações necessárias para a construção da árvore da mensagem de exemplo (2/3).

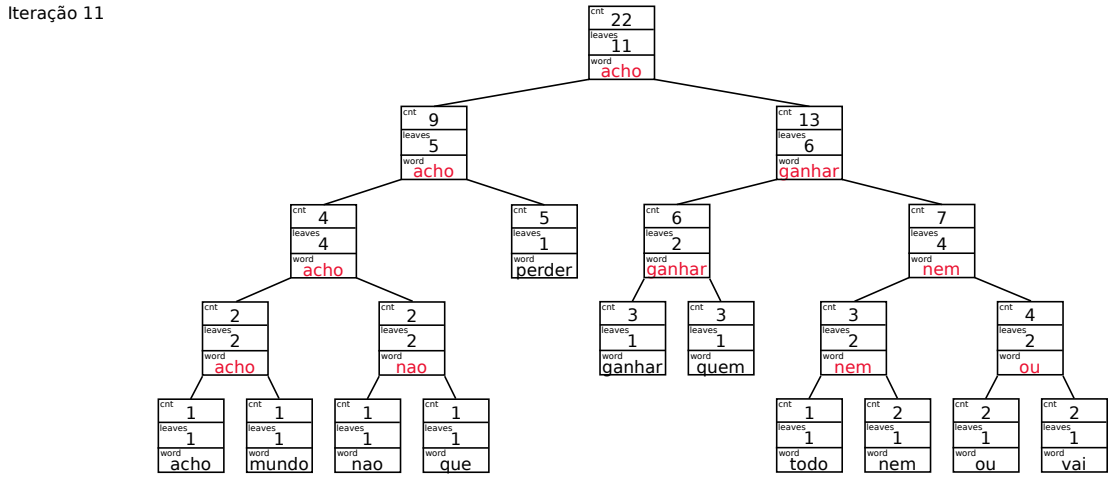

Figura 6: Iterações necessárias para a construção da árvore da mensagem de exemplo (3/3).

Distribuição dos Pontos

Contagem de Palavras	Código Huffman	Boas Práticas	Documentação	Total
3.5	3.5	1.0	2.0	**10.0**

Entregáveis

Código-fonte. A implementação poderá ser feita utilizando as linguagens C ou C++. Não será permitido o uso da *Standard Library* do C++ ou de bibliotecas externas que implementem as estruturas de dados ou os algoritmos. **A implementação das estruturas e algoritmos utilizados neste trabalho deve ser sua. É expressamente proibido utilizar estruturas de busca prontas**. Os códigos devem ser executáveis em um computador com *Linux*. Caso não possua um computador com *Linux*, teste seu trabalho em um dos computadores do laboratório de graduação do CRC[2]. A utilização de *Makefile*[3] é **obrigatória** para este trabalho.

Aplique boas práticas de programação e organize seu código-fonte em arquivos, classes e funções de acordo com o significado de cada parte. A separação de responsabilidades é um dos princípios da engenharia de software: cada função deve realizar apenas uma tarefa e cada classe deve conter apenas métodos condizentes com sua semântica.

Documentação. A documentação de seu programa **deverá** estar em formato **PDF**, **seguir** exclusivamente o modelo de trabalhos acadêmicos da SBC (que pode ser encontrado online[4]) e ser **sucinta**, não excedendo o limite de 7 páginas. Deverá conter **todos** os seguintes tópicos:

- Cabeçalho. Título do trabalho, nome e número de matrícula do autor.
- Introdução. Apresentação do problema abordado e visão geral sobre o funcionamento do programa.

[2] `https://crc.dcc.ufmg.br/infraestrutura/laboratorios/linux`
[3] `https://opensource.com/article/18/8/what-how-makefile`
[4] `http://www.sbc.org.br/documentos-da-sbc/summary/169-templates-para-artigos-e-capitulos-de-livros/878-modelosparapublicaodeartigos`

- Implementação. Descrição sobre a implementação do programa. Devem ser detalhados o funcionamento das principais funções e procedimentos utilizados, o formato de entrada e saída de dados, compilador utilizado, bem como decisões tomadas relativas aos casos e detalhes que porventura estejam omissos no enunciado. **Não devem ser inseridos trechos de código fonte na documentação**.
- Instruções de compilação e execução. Instruções de como compilar e executar o programa.
- Análise de complexidade. Estudo da complexidade de tempo e espaço do algoritmo de melhor e pior caso desenvolvido utilizando o formalismo da notação assintótica.
- Conclusão. Resumo do trabalho realizado, conclusões gerais sobre os resultados e eventuais dificuldades ou considerações sobre seu desenvolvimento.
- Bibliografia. Fontes consultadas para realização do trabalho.

O código-fonte e a documentação devem ser organizados como demonstrado pela árvore de diretórios na Figura 7. O diretório raiz deve ser nomeado de acordo seu **nome e último sobrenome**, separado por *underscore*, por exemplo, o trabalho de "Kristoff das Neves Björgman" seria entregue em um diretório chamado `kristoff_bjorgman`. Este diretório principal deverá conter um subdiretório chamado `src`, que por sua vez conterá os códigos (`.cpp, .c, .h, .hpp`) na estrutura de diretórios desejada. A documentação **em formato PDF** deverá ser incluída no diretório raiz do trabalho. Evite o uso de caracteres especiais, acentos e espaços na nomeação de arquivos e diretórios.

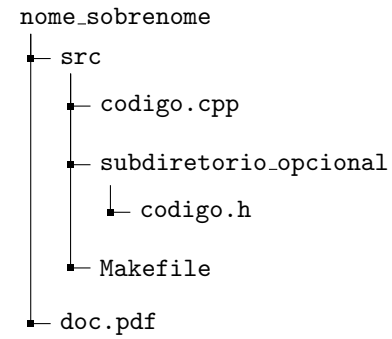

Figura 7: Estrutura de diretórios do entregável do TP3

O diretório deverá ser submetido em um único arquivo **'nome_sobrenome.zip'**, (onde nome e sobrenome seguem as mesmas diretrizes para o nome do diretório, explicado acima) através do Moodle da disciplina até as **23:59** do dia **27 de novembro de 2019**.

Considerações Finais

Algumas considerações finais importantes:

- **Preste bastante atenção nos detalhes da especificação**. Cada detalhe ignorado acarretará em perda de pontos.
- O que será avaliado no trabalho:

  Boas práticas de programação: se o está código bem organizado e indentado, com comentários explicativos, possui variáveis com nomes intuitivos, modularizado, etc.

  Implementação correta dos algoritmos: se as estruturas e funções foram implementadas de forma correta e resolvem o problema aqui descrito.

Conteúdo da documentação: se todo o conteúdo necessário está presente, reflete o que foi implementado e está escrito de forma coerente e coesa.

- Após submeter no Moodle seu arquivo **'.zip'**, faça o download dele e certifique-se que não está corrompido. Não será dada segunda chance de submissão para arquivos corrompidos.
- Em caso de dúvidas, **não hesite em perguntar** no Fórum de Discussão no Moodle ou procurar os monitores da disciplina – estamos aqui para ajudar!
- **PLÁGIO É CRIME**: caso utilize códigos disponíveis online ou em livros, **referencie** (inclua comentários no código fonte e descreva a situação na documentação). Trabalhos onde o plágio for identificado serão devidamente penalizados: o aluno terá seu trabalho anulado e as devidas providências administrativas serão tomadas. Discussões sobre o trabalho entre colegas são encorajadas, porém compartilhamento de código ou texto é plágio e as regras acima também se aplicam.
- Em caso de atraso na entrega, serão descontados $2^d - 1$ pontos, onde d é o número de dias (corridos) de atraso arredondado para cima.
- Comece o trabalho o mais cedo possível. Você nunca terá tanto tempo pra fazê-lo se começar agora!

Bom trabalho!

Apêndices

A Dicas para a documentação

O objetivo desta seção é apresentar algumas dicas para auxiliar na redação da documentação do trabalho prático.

1. **Sobre *Screenshots***: ao incluir *screenshots* (imagens da tela) em sua documentação, evite utilizar o fundo escuro. Muitas pessoas preferem imprimir documentos para lê-los e imagens com fundo preto dificultam a impressão e visualização. Recomenda-se o uso de fundo branco com caracteres pretos, para *screenshots*. Evite incluir trechos de código e/ou pseudocódigos utilizando *screenshots*. Leia abaixo algumas dicas de como incluir código e pseudocódigo em seu texto.

2. **Sobre URLs e referências**: evite utilizar URLs da internet como referências. Geralmente URLs são incluídas como notas de rodapé. Para isto basta utilizar o comando `\footnote{\url{}}` no LaTeX, ou ativar a opção nota de rodapé[5] no Google Docs/MS Word.

3. **Evite o Ctrl+C/Ctrl+V**: encoraja-se a modularização de código, porém a documentação é única e só serve para um trabalho prático. Reuso de documentação é auto-plágio[6]!

[5] `https://support.office.com/en-ie/article/insert-footnotes-and-endnotes-61f3fb1a-4717-414c-9a8f-015a5f3ff4cb`
[6] `https://blog.scielo.org/blog/2013/11/11/etica-editorial-e-o-problema-do-autoplagio/#.XORgbdtKg5k`