Trabalho Prático 3: Decifrando os Segredos de Arendelle

Victor Hugo Silva Moura

2018054958

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

victorhugosmoura@gmail.com

Abstract. *The following text describes the "Deciphering Arendelle's Secrets" problem and its solution using a variation of the Binary Trie. This Trie is used to store the Morse codes and its respective representation, making the recovery of information easier.*

Resumo. *O texto a seguir descreve o problema "Decifrando os Segredos de Arendelle" e sua resolução usando uma variação da Trie Binária. Essa Trie é utilizada para armazenar os códigos Morse e suas respectivas representações, facilitando, assim, a recuperação da informação.*

1. Introdução

O problema *Decifrando os Segredos de Arrendelle* consiste em decifrar mensagens em códigos Morse utilizados, no passado, pela realeza de Arendelle. Para manter os segredos da família real ainda mais seguros, foi utilizada uma variação do código Morse. Sendo assim, para resolver esse problema é necessário saber a representação dos símbolos na variação utilizada e armazená-los de forma que seja fácil recuperá-los posteriormente.

O armazenamento dos símbolos e de seus códigos é feito por meio de uma variação da Trie Binária (será detalhado na seção 2). A escolha dessa estrutura se deve ao modo pelo qual os dados são dispostos na árvore. Essa disposição facilita a recuperação deles posteriormente, algo bem importante para decifrar as mensagens de forma rápida. A linguagem utilizada na solução desse problema foi a linguagem C.

2. Implementação

A implementação desse problema se deu em duas etapas distintas, sendo elas:

- Construção da Trie Binária utilizando os códigos Morse e símbolos fornecidos por um arquivo de entrada;
- Utilização da Trie para decifrar as mensagens em Morse e retorná-las para o usuário.

Sendo assim, vamos dividir essa seção em duas partes para abordar cada etapa individualmente.

2.1. Construção da árvore

Para construir a Trie Binária que armazenará os dados (código Morse e caractere correspondente ao código) é necessário antes saber quais são eles. Os dados para a construção da árvore são passados por meio de um arquivo de texto *morse.txt*. Esse arquivo contém em cada linha o símbolo (caractere) e seu código Morse,

respectivamente. Cada código é então colocado na árvore seguindo o critério: começando a partir da raiz, para cada caractere do código, se este for um ponto '.' a posição do dado na árvore está na subárvore filha esquerda do nó atual. Caso o caractere seja um traço '-', a posição do dado está na subárvore filha direita do nó atual. Após todos os caracteres do código Morse serem lidos, o dado estará em sua posição final na árvore e será inserido nessa posição.

O que esse processo faz, em outras palavras, é procurar a exata posição do dado na árvore e colocá-lo lá, seguindo uma lógica parecida com a da árvore binária de pesquisa (porém, em vez de considerar menor e maior para organizar os dados, consideramos pontos e traços do código Morse). Como os dados não estão ordenados no arquivo por modo de inserção na árvore, alguns nós de apoio são criados durante o processo de busca da posição do nó. Esses nós de apoio são nós vazios, que podem ser utilizados posteriormente caso haja um dado na posição deles, e que servem apenas para direcionar o dado atual a ser inserido para sua posição correta. A figura 1 demonstra um exemplo da Trie gerada para os caracteres *E, U, F, T, G, O*. Todos os nós com o símbolo Ø são nós de apoio, com exceção da raiz.

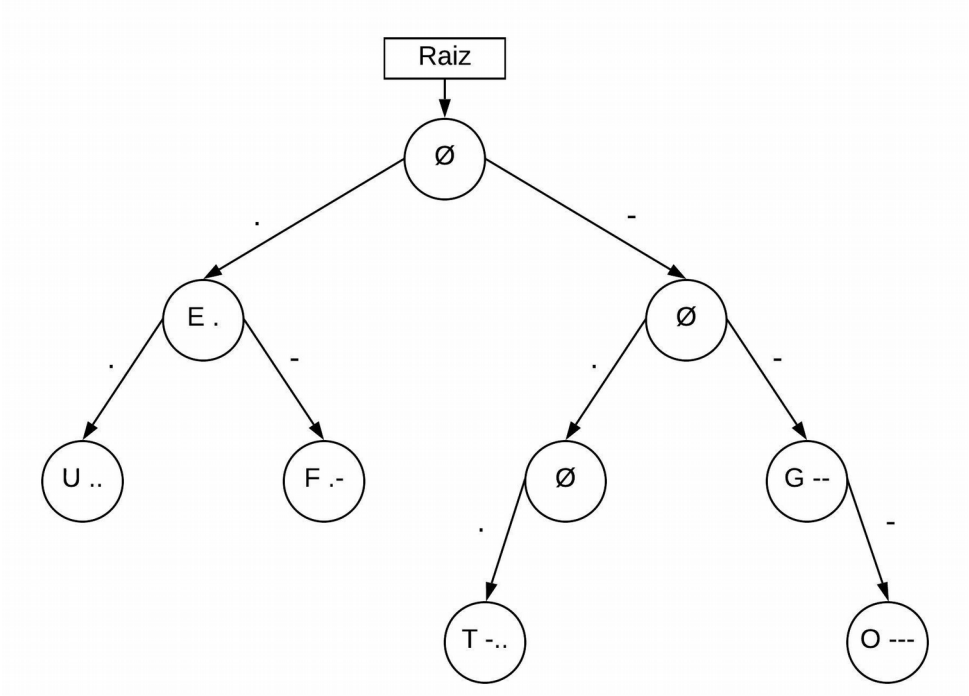

Figura 1. Exemplo de Trie para os caracteres E, U, F, T, G e O

Esse processo é feito pela função *insertR*, função recursiva chamada pela função *insert*. Essa função recebe um ponteiro para algum nó da trie, o dado a ser inserido e um inteiro que marca qual caractere do código Morse está sendo lido no momento, e retorna um ponteiro para o nó atual, seja ele um nó de apoio ou o nó onde o dado será inserido. Esse retorno de ponteiros é importante para que a estrutura de apontadores da árvore seja mantida.

2.2. Conversão das Mensagens

Após montar a árvore para armazenar os dados, é necessário utilizá-la para decodificar as mensagens em Morse inseridas pelo usuário. A decodificação das mensagens depende de duas etapas: saber a formatação da entrada para identificar corretamente os

códigos Morse e, após extrair um código da entrada, procurar pelo caractere correspondente a ele na árvore.

A busca pelo caractere funciona de forma parecida com a inserção, com a diferença de que não é necessário criar nenhum nó durante esse processo. A pesquisa pode ser descrita como: começando da raiz da árvore, para cada caractere do código Morse, se este for um ponto '.', pesquisar na subárvore filha esquerda do nó atual. Caso seja um traço '-', pesquisar na subárvore filha direita do nó atual. Quando todos os caracteres do código forem lidos, verificar se o nó existe na posição e se é o nó correto. Se sim, retorna um ponteiro para ele. Se não, retorna um ponteiro para NULL. Caso em algum dos passos de pesquisa, a próxima subárvore a ser pesquisada não exista, retorna-se um ponteiro para NULL indicando que o nó não foi encontrado.

Em outras palavras, a pesquisa procura a exata posição do nó na árvore do mesmo jeito que a inserção, porém caso não exista o caminho exato até o nó ou caso o nó seja diferente do que se deseja, é considerado que o nó não existe. Isso é feito pela função *searchR*, função recursiva chamada pela função *search*. Ela recebe um ponteiro para uma posição da árvore, o dado a ser buscado e um inteiro que marca qual caractere do código Morse está sendo lido no momento, e retorna um ponteiro para o nó a ser buscado.

Para a leitura de dados, a formatação da entrada segue o seguinte padrão:

.-.. ..- …- .-. . / . … -.. --- …

onde cada bloco de pontos e traços representa um código Morse e as barras '/' representam o espaço em uma frase. Cada código é separado dos outros por um espaço, bem como as barras.

Cada linha da entrada é lida como um único string e os códigos Morse são extraídos por meio da função *strtok*. Essa função retorna uma substring da string principal limitada por um token. Por exemplo, se a string principal é "Hora de Aventura" e o token passado para a função é um espaço ' ', ela ira retornar a substring "Hora". Caso a função seja chamada novamente, porém com o parâmetro da string NULL, ela retorna a próxima substring até o token. No exemplo, retornaria a substring "de". Com essa separação, basta usar um espaço como token e usar o método até o fim da string.

A decodificação fica guardada em uma string diferente da string de entrada de forma a evitar um possível conflito de dados. Para cada substring gerada a partir da entrada, se ela for uma barra, adiciona-se um espaço na string que guarda a mensagem decodificada. Se ela for um código morse, é feita uma busca por esse código na árvore e o resultado é inserido na string. Ao chegar ao final da string de entrada, a string que guarda a mensagem decodificada é armazenada em uma fila de strings e é reinicializada para receber uma nova mensagem. Esse processo é finalizado quando a entrada indica um fim de arquivo (EOF).

Ao final da leitura da entrada, a fila de strings é impressa revelando as mensagens decodificadas. Como o número de linhas da entrada é desconhecido e deseja-se manter a ordem das linhas a escolha de uma estrutura de dados do tipo fila se mostrou mais coerente.

Além de todo esse processo, existe um parâmetro adicional '-a' que pode ser adicionado ao programa da seguinte forma:

./nomedoprograma -a

Caso esse parâmetro esteja presente na chamada do programa, a Trie Binária gerada pelo programa deverá ser impressa (caractere seguido do código Morse) após as decodificações, seguindo o caminhamento pré-ordem. Os nós vazios não são impressos nesse processo. Para o exemplo da figura 1, o caminhamento seria *E, U, F, T, G e O*.

3. Instruções de Compilação e Execução

O compilador utilizado durante toda a implementação foi o GNU C Compiler (GCC) versão 7.3.0 e o sistema operacional foi um Linux Ubuntu 18.04 64-bit.

Para compilar o programa foi desenvolvido um arquivo makefile que é responsável por gerenciar os arquivos a serem compilados e a ordem de compilação. Para executar esse aqui, basta digitar o comando *make* no terminal.

A execução do programa é feita pelo comando:

./nomedoprograma [-a]

4. Análise de Complexidade

O computador utilizado para a implementação segue as seguintes especificações técnicas:

- **Processador:** Intel i7-3612QM CPU @ 2.10GHz
- **Memória RAM:** 4 GB
- **Sistema Operacional:** Linux Ubuntu (versão 18.04)

4.1. Complexidade de Tempo

Ao analisar a complexidade de tempo, é possível dividi-la da mesma forma que a implementação. Sendo assim, vamos começar analisando a complexidade de tempo envolvida na criação da árvore e posteriormente na decodificação da mensagem.

A árvore a ser criada é sempre a mesma (mesmos caracteres na mesma ordem já preestabelecida pelo arquivo *morse.txt*). Dessa forma podemos considerar o custo de criação da árvore como constante, já que o processo envolvido será executado da mesma forma para todo e qualquer caso de testes. Então a complexidade de tempo para criação da árvore é *O(1)*.

Para decodificar uma mensagem temos o custo de pesquisar um código na árvore para cada caractere. Utilizando da mesma lógica anterior, a pesquisa é constante, dado que a árvore nunca se altera de acordo com a entrada. Sendo assim, uma pesquisa de um caractere é *O(1)*. Se consideramos o tamanho total da mensagem como n, teremos um custo de $n * O(1) = O(n)$ que é o custo total para decodificar uma mensagem de n caracteres.

Pode-se considerar também o custo de impressão da mensagem e da árvore (caso necessário). Porém estes são mais fáceis de serem analisado. Cada caractere da mensagem terá que ser impresso, o que dá um custo total *O(n)*. Para a árvore, como sempre será a mesma impressão, teremos um custo constante *O(1)*.

Com isso, a complexidade de tempo total do programa é

$$O(1) + O(n) + O(n) + O(1) = O(n)$$

4.2. Complexidade de Espaço

Para a complexidade de espaço vamos analisar o espaço utilizado para armazenar a árvore e para armazenar as mensagens decodificadas.

A árvore, como dito na complexidade anterior, tem tamanho fixo determinado pelo arquivo de entrada. Sendo assim, o tamanho dela sempre será o mesmo em todos os casos de teste, ou seja, um tamanho constante. Assim, a complexidade de espaço para armazenar a árvore é $O(1)$.

Para armazenar as mensagens, vamos começar "cortando" alguns custos que são constantes e não impactarão na complexidade final. É utilizada apenas uma string para pegar um código morse a ser decodificado, uma string de entrada e uma auxiliar que guarda a mensagem decodificada antes de ser inserida na fila de strings. Todas essas strings são criadas apenas uma vez e os dados são sobrescritos nelas à medida que necessário, o que torna o custo delas constante.

Considerando que o usuário entre com uma mensagem correspondente a n caracteres (decodificados) no programa, todos eles serão armazenados na fila de strings, em alguma das strings. Sendo assim, o custo total para armazenar esses caracteres é da ordem do número de caracteres da saída final, ou seja, $O(n)$.

Com isso, a complexidade de espaço total do programa é

$$O(1) + O(n) = O(n)$$

5. Conclusão

Por meio de uma implementação simples, a realização desse trabalho mostrou uma forma alternativa de armazenar e buscar dados utilizando uma estrutura de árvore, já estudada anteriormente, porém com uma organização diferente dos dados. Os casos utilizados para teste são pequenos, mas a análise de complexidade mostra que a estrutura de dados utilizada na resolução é bem eficiente se comparada ao método de busca sequencial, por exemplo. Apesar de pequenas dificuldades durante a implementação da solução, o desenvolvimento dela foi bem simples e objetivo.

6. Referências

Ziviani, N. *Projeto de Algoritmos: com implementações em PASCAL e C. 2 ed.* São Paulo: Thomson, 2004

Chaimowicz, L. e Prates, R. *Pesquisa Digital*, disponível na turma *ESTRUTURAS DE DADOS – METATURMA* do Moodle UFMG 2019/1. Acesso em julho/2019.

Ahuja, K. e Patade, A. *Command line arguments in C/C++,* https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/, acesso em julho/2019.

Autor Desconhecido. *C Reference*. http://www.cplusplus.com, acesso em julho/2019.