Trabalho Prático 3
Decifrando os Segredos de Arendelle

Breno Poggiali de Sousa
brenopoggiali@ufmg.br
2018054800

Universidade Federal de Minas Gerais (UFMG)

1. Introdução

Decifrando os Segredos de Arendelle surgiu após o seguinte problema no reino de Arendelle: o Serviço Secreto Arendellense (SSA) possuía uma grande quantidade de arquivos criptografados por um código Morse. E, como a maioria dos profissionais peritos em código Morse no reino haviam sido desligados em regime de aposentadoria voluntária. Nos chamaram para tentar solucionar esse problema. A ideia principal é usar uma árvore binária onde, a cada "passo" para a esquerda, seria como se fosse um ponto - "." e , cada "passo" para a direita seria como se fosse um "-" no código morse.

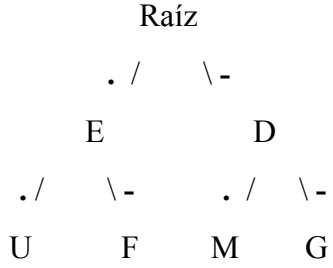
2. Implementação

Toda a implementação - excluindo a que está extra no Apêndice A desse documento - foi desenvolvida utilizando a linguagem C++ e compilada pelo GNU Compiller Collection.

2.1 Estrutura de dados

A estrutura de dados utilizada para desenvolver esse trabalho foi a de uma árvore binária. Mais especificamente, um caso particular dela onde, não precisamos fazer comparações entre as chaves de cada nó. E sim, recebemos uma chave com um número binário (ou no nosso caso, código morse que só contêm "." ou "-"). E, determinamos que cada um vai para um lado. Isso nos permite ir caminhando pela árvore de forma que, no final, para cada código (não existem códigos que repetem entre si), só teremos uma única chave e poderemos gravar ela na posição adequada na árvore.

A árvore binária construída segue a implementação demonstrada abaixo:



Basicamente, iremos construir a árvore primeiro, baseado em um arquivo .txt que contém a chave e os códigos para cada chave. E, a partir daí, já podemos decifrar qualquer arquivo escrito nessa variação de código morse!

2.2 Classes

Neste trabalho, optamos por uma abordagem simplificada que resolvesse o problema de forma eficiente. Por isso, utilizamos apenas uma classe que será descrita logo abaixo.

2.2.1 Árvore.cpp

Essa classe é o coração do nosso trabalho. É nela que realizamos tanto a montagem da árvore-dicionário, que chamamos de *AlphabetMorseTree*, quanto as pesquisas de impressão e a própria impressão da árvore na pré ordem, assim como solicitado pelo governo de Arendelle.

Nessa classe, possuímos 8 funções além de um struct. O usado se trata de um nó (node) que possui 4 variáveis, sendo 2 apontadores para nós do seu lado direito e esquerdo, 1 para armazenar a chave principal, que chamamos de *value* e representa o caractere que será retornado e outra para guardar a chave que libera esse *value*, que também precisaremos quando o parâmetro -a for passado.

Já quando as funções, são as 8:

***AlphabetMorseTree()*:** construtor da classe que cria o primeiro nó e atribui à raíz da árvore e chama a função *CreateAlphabetTree()* que construirá todo o resto da árvore.

***tree_destroyer(node*)*:** destruidor de nós recursivo.

***~AlphabetMorseTree()*:** destruidor da classe que chama a função *tree_destroyer(node*)* passando a raíz da árvore como parâmetro.

CreateAlphabetTree(): construtor da árvore que lê o arquivo “morse.txt” e chama os inserts para inserir todos os *value's* e *code's* na árvore.

insert(char, string): cria os nós restantes necessários e adiciona o *value* e o *code* de cada código no lugar correto.

search(string): recebe um código como parâmetro e percorre a árvore por cada elemento deste código até acabar e chegar no nó onde está armazenado o valor que seria a tradução do código morse para ascii.

print_preorder(node*): recebe um nó e imprime o caminhamento pré-ordem desse nó recursivamente.

print_tree(): chama a função *print_preorder(node*)* passando a raíz como nó.

3. Instruções de compilação e execução

Para executar o programa, você precisa compilá-lo primeiro.

3.1 Compilar

Como estamos utilizando um *Makefile* e nos foi solicitado que armazenássemos o *Makefile* dentro da pasta src, para compilar-lo, a partir da pasta principal, apenas precisamos rodar:

```
$ cd src
$ make
```

e, caso queira rodar os testes, após isso deve ser executado:

```
$ make tests
```

Nesse último caso, caso haja alguma discrepância entre os valores dos testes e o executado pelo programa, o make tests vai lhe mostrar. Caso ele não mostre nada, é porque tudo fluiu como esperado!

3.2 Execução

Após compilado, o programa poderá ser executado em um terminal de um computador Linux com o comando:

```
$ ./main
```

e, inserindo manualmente a entrada e visualizando a saída no próprio terminal. Nesse caso, você pode escrever a vontade e terminar de executar a leitura de código morse ou com o padrão "**Ctrl+D**" ou então digitar "**fim**". Outra opção é executar o programa com:

```
$ ./main < exemplo.in > exemplo.out
```

Efetuando a leitura da entrada contida no arquivo exemplo.in e escrevendo a saída no arquivo exemplo.out.

Ou ainda:

```
$ ./main -a < exemplo.in > exemplo.out
```

Também efetuando a leitura da entrada contida no arquivo exemplo.in mas escrevendo a saída no arquivo exemplo.out com adição do caminhamento em pré-ordem da árvore de pesquisa.

4. Análise de Complexidade

A complexidade desse trabalho é relativamente baixa. Principalmente porque o tamanho da nossa árvore é constante e baixo. Por se tratar de uma BST, a simples consulta/inserção, possui complexidade O(log n), já que estamos sempre diminuindo nossas opções pela metade, em média. Já quanto aos códigos, para salvar as sequências, temos que iterar de 1 por 1 e, por isso, possuímos O(n).

Por fim, na consulta na árvore, como já dissemos, por ser uma BST, estamos sempre dividindo na metade e temos um caso médio igual a O(log n). Por favor, dê uma olhada no **Apêndice A** e entenda porque utilizar dicionários (em Python) faz essa etapa ter O(1) para cada caso.

5. Conclusão

Portanto, conseguimos perceber nesse trabalho que o trabalho de decodificação de um código morse é um processo extremamente repetitivo e que, é facilmente mapeado para ser realizado/decodificado por uma máquina. Ademais, percebemos também que uma BST pode ser uma implementação bem eficiente para casos em que temos situações binárias (0 ou 1, direita ou esquerda, “.” ou “-”). Por fim, observando o Apêndice A, vimos também que existem implementações de solução desse problema mais eficientes e menos custosas utilizando um dicionário (e que bem provavelmente podem ser do interesse da Rainha de Arendelle).

6. Referências

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em PASCAL e C: Capítulo 5: Pesquisa em Memória Primária. Editora Cengage.

Apêndice A

No código em Python abaixo, resolvemos o mesmo problema de uma forma mais eficiente do que a proposta. Utilizando dicionários. O interessante dessa estrutura é que ela possui O(1) para pesquisa. Isso porque ela funciona como um array, onde a chave é o "i-ésimo" do array. Analogamente, seria como se tivéssemos um array onde poderíamos pesquisar: array['nota_breno_tp3'] e ela já iria retornar diretamente 10. Como todos sabemos, num array mesmo, só conseguimos passar números que determinam posições. Segue o código:

```
def decodificate(morse_code):
  letter_in_morse = ''
  for line in morse_code:
    for bip in line:
      if letter_in_morse == '/':
        print(' ', end='')
        letter_in_morse = ''
      elif bip == ' ':
        print(morse_dict[letter_in_morse], end='')
        letter_in_morse = ''
      else:
        letter_in_morse += bip
    print(morse_dict[letter_in_morse])
    letter_in_morse = ''

def create_morse_dict(path):
  morse_file = open(path, 'r')
  char, sequence = '', ''
  created_morse_dict = {}
  for line in morse_file:
    for i in range(len(line)):
      bip = line[i]
      if i == 0:
        char = bip
      elif bip in {'.', '-'}:
        sequence += bip
    created_morse_dict[sequence] = char
    sequence = ''
  return created_morse_dict
```

```
# "MAIN"
morse_input = []
while True:
    try:
        line = input()
    except EOFError:
        break
    morse_input.append(line)

morse_dict = create_morse_dict('./src/morse.txt')
decodificate(morse_input)
```