UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Documentação Trabalho Prático 3
Estruturas de Dados - DCC205 - TF

Um Resgate Inesperado

Luiz Gustavo Almeida de Oliveira
Matrícula: 2019083781
luizorst@hotmail.com

29 de março de 2021

Sumário

1 Introdução **2**
1.1 Contextualização 2

2 Instruções **3**
2.1 Compilação e Execução 3
2.1.1 Gerando o executável sem o programa *GNU Make* 3
2.1.2 Gerando o executável usando o programa *GNU Make* 3

3 Análise do código **4**
3.1 TranslationUnit 4
3.1.1 Construção da Árvore | buildTreeFromString 4
3.1.2 Decodificação de uma Mensagem | decodeMessage 4
3.1.3 Codificação de uma Mensagem | encodeMessage 5

4 Conclusão **6**

1 Introdução

1.1 Contextualização

Com a definição das bases prioritárias, o plano para trazer todo o Composto-Z que foi extraído pelo Universo de volta para a Terra seguiu, porém um novo empecilho surgiu, piratas mercenários. Durante a viagem de retorno, as naves tiveram a comunicação com a central de controle bloqueada e ficaram presas no espaço profundo, dando aos piradas o tempo que precisam para roubar o carregamento de Composto-Z.

Diante dessa enrascada, a missão reestabeleceu a comunicação através de um novo canal emergencial. Esse canal serve para as naves se comunicarem com o centro de controle e, como medida de segurança, mostrou-se necessário codificar as mensagens enviadas. Cada letra é representada por um código alfanumérico que decodifica o caminho a ser percorrido numa árvore binária. Apenas a central conhece a árvore que foi utilizada para codificar as mensagens de cada nave.

Para a implementação da lógica de comunicação descrita, conforme mencionado anteriormente, foi utilizada uma árvore binária de busca, sendo que para decodificar mensagens navegamos na mesma através de um códido numérico e, para codificar, utilizamos o caminho até o caractere na árvore para gerar um número aleatório correspondente àquele caminho.

2 Instruções

2.1 Compilação e Execução

Pré-requisitos:

- Compilador moderno de C++(e.g: g++ 10.x.x)
- Programa *GNU Make* instalado (altamente recomendado)

Primeiro, deve-se extrair os arquivos em um diretório onde deseja que os mesmos fiquem armazenados. As instruções para gerar o binário com ou sem o programa *GNU Make* estão nas próximas subseções, mas considerando que ele foi gerado com sucesso, ele deve ser utilizado da seguinte maneira, assumindo que você está na mesma pasta que o executável e que o arquivo contendo a entrada está no mesmo diretório também:

```
./meu_executavel [Caminho arq. com dados]
```

O executável gerado pelo Makefile tem o nome padrão run.out, então, exemplificando novamente, assumindo que o arquivo entrada.txt está no mesmo diretório que o executável, basta:

```
./run.out entrada.txt
```

2.1.1 Gerando o executável sem o programa *GNU Make*

Caso o programa *GNU Make* não esteja instalado, será necessário entrar na pasta *Projeto* e compilar manualmente os arquivos, fazendo a inclusão dos arquivos na pasta *include* no comando, por exemplo:

```
// pwd output: ./mydir/Projeto
g++ -I ./include/*.hpp src/*.cc
```

2.1.2 Gerando o executável usando o programa *GNU Make*

O arquivo comprimido contém, também, um *Makefile*, que facilita a construção do executável. Com o programa *GNU Make* instalado, considerando que o arquivo comprimido original foi extraído em uma pasta qualquer, é necessário apenas entrar na pasta *Projeto* e digitar o comando *make*. Depois disso, será gerado um binário na pasta *bin*, que deve ser usado para executar o programa.

3 Análise do código

Para a implementação da solução proposta para o problema de comunicação, foi necessário não muito mais do que uma árvore binária de busca com alguns métodos específicos para, por exemplo, gerar um caminho para um certo nó dela. A função *main* do programa tem estrutura simples e consiste na leitura dos comandos do arquivo indicado e então execução das ordens, imprimindo algo na saída padrão quando necessário.

3.1 TranslationUnit

Esse é o principal componente do programa, ele contém a árvore binária de pesquisa e os métodos responsáveis por tratar as mensagens(codificação e decodificação). Os seus procedimentos da interface pública da classe serão estudados à seguir.

3.1.1 Construção da Árvore | buildTreeFromString

Antes de codificar ou decodificar qualquer mensagem, é necessário que já exista uma árvore binária de pesquisa montada dentro da instância, pois só assim será possível navegar para encontrar um caractere e realizar os procedimentos necessários. Sabendo que estamos lidando com uma árvore binária, a complexidade assintótica média para a construção de uma árvore é $O(n \cdot \log n)$, pois fazemos a inserção, que é $O(\log n)$, n vezes.

Em questão da complexidade espacial da árvore, ela cresce conforme o número de nós aumenta, de maneira linear e diretamento proporcional à entrada de tamanho n, tendo, portanto, complexidade assintótica espacial $O(n)$.

Complexidade	Temporal	Espacial
	$O(n \cdot \log n)$	$O(n)$

3.1.2 Decodificação de uma Mensagem | decodeMessage

Quando uma mensagem precisa ser decodificada, o método *decodeMessage* é chamado, recebendo o conteúdo à ser decodificado como argumento. Dentro dele, o procedimento consiste em navegar na mensagem até encontrar o caractere que marca o início da próxima letra a ser decodificada, chamando então o método *decodingSearchFromStringProcedure* com a substring que corresponde ao caminho para a letra correspondenten na árvore. Esses procedimentos são realizados até o conteúdo da mensagem ter sido decodificado completamente e então o mesmo é retornado.

Existe um *while loop* responsável por continuar buscando o caractere que marca o início de um novo caminho para um caractere na árvore, essa checagem será feita um número de vezes que tende a crescer linearmente com o tamanho da sting recebida de entrada. Dentro desse loop, o método que realiza a busca na árvore é chamado, tendo complexidade temporal $O(\log n)$, já que a cada nó filho escolhido para continuar a busca, metade da árvore do pai é descartada. Tendo em vista o crescimento linear de vezes que o *while loop* será executado e a natureza logarítmica do procedimento que acontece dentro

dele, essa parte da decodificação domina seu tempo de execução e tem complexidade assintótica temporal $O(n \cdot \log n)$.

Tratando da complexidade espacial, já que cada caractere decodificado é colocado na *stringstream* da mensagem decodificada que posteriormente será retornada, conforme a entrada cresce, o número de caracteres cresce e, consequentemente, o espaço necessário para decodificação também, portanto a complexidade assintótica temporal é $O(n)$.

Complexidade	Temporal	Espacial
	$O(n \cdot \log n)$	$O(n)$

3.1.3 Codificação de uma Mensagem | encodeMessage

Tratando da codificação de mensagens, o procedimento é bastante similar ao explicado anteriormente. É necessário navegar a árvore por meio da entrada mas, ao invés de gerar uma saída decodificada, vamos gerar uma saída codificada usando o caminho percorrido até encontrar um certo caractere.

A função *encodeMessage* itera sobre os caracteres da mensagem que precisa ser codificada e, para cada um deles, chama o método *numberPathFromChar*, que busca pelo caractere na árvore e então gera um número aleatório que, por meio do método de decodificação, se usado na mesma árvore, leva ao caractere. O método que gera o número aleatório correspondente ao caminho para o caractere consiste na chamada de um método da árvore binária de busca que retorna uma string onde caractere corresponde à uma movimentação para o nó da direita ou da esquerda, através dessa string é possível gerar números pares, que correspondem à uma ida para a direta, ou ímpares, que correspondem à uma ida para a esquerda.

Portanto, para cada letra da mensagem que precisa ser codificada, fazemos uma busca pelo caractere na árvore($O(\log n)$ e, posteriormente, "traduzimos"a string para números gerados aleatoriamente enquanto iteramos sobre ela, sendo que a string do caminho tem o tamanho máximo de $\log n$). Então, é possível concluir que a complexidade assintótica temporal é $O(n \cdot (\log n)^2)$

Tratando da complexidade espacial, ficamos no mesmo caso que o método de decodificação, já que a mensagem que está sendo codificada vai sendo salva e, conforme o tamanho da entrada do que deve ser codificado aumenta, maior é o espaço necessário para armazenar a mensagem codificada, levando à complexidade assintótica temporal $O(n)$.

Complexidade	Temporal	Espacial
	$O(n \cdot (\log n)^2)$	$O(n)$

4 Conclusão

Diante de tudo que foi exposto, é possível concluir que agora as espaço naves terão uma alternativa segura para comunicação utilizando uma solução baseada nas árvores binárias de busca.

Através da análise da complexidade assintótica temporal e espacial do programa, foi possível mostrar que são dominadas por:

Complexidade	Temporal	Espacial
	$O(n \cdot (\log n)^2)$	$O(n)$

Por fim, nesse trabalho foi possível verificar a validade de diversos conceitos estudados na disciplina de Estruturas de Dados[1]. Em especial, a análise de complexidade do código se mostrou bastante útil na hora de fazer escolhas sobre como certos procedimentos deveriam ser implementados, além de praticar a implementação de uma árvore binária de busca em um contexto de uma aplicação, sendo que essa estrutura permitiu fazer uma implementação mais eficiente para os métodos de busca.

Referências

[1] L. Chaimowicz and R. Prates, “Slides virtuais da disciplina de Estruturas de Dados,” disponibilizado via moodle. Departamento de Ciencia da Computacao. Universidade Federal de Minas Gerais. Belo Horizonte.