UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Documentação Trabalho Prático 1
Estruturas de Dados - DCC205 - TF

Extração - Z

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

14 de fevereiro de 2021

Sumário

1 Introdução

Com o colapso da sociedade como é conhecida no sécula XXI, em um futuro distante, os avanços tecnológicos do Instituto de Ciências Exoplanetárias da Terra permitiram que mais uma vez as pessoas possam sonhar. A razão por trás disso é a descoberta de um novo composto, Composto Z, que poderá ser usado para realizar um processo de terraformação e retornar a Terra para um estado de ampla biodiversidade.

Essa documentação tem como objetivo descrever de maneira geral o programa feito para gerenciamento dos robôs e coordenação das operações de extração do Composto Z no planeta alvo. Além disso, serão explicadas as principais escolhas de implementação e o custo temporal e espacial do programa e de seus procedimentos mais relevantes.

De maneira breve, pode-se dizer que o programa foi implementado seguindo os paradigmas de desenvolvimento orientado por testes e de orientação à objeto. Inicialmente, depois de estudar o problema que deveria ser tratado, foram implementadas as estruturas necessárias para organizar as informações, tais como uma fila e uma lista, para manter as ordens de comando e os robôs da base, respectivamente. Depois da implementação das estruturas iniciais, novas classes referentes à natureza do problema foram implementadas, como a classe Robô, Mapa e Base. Por fim, fazendo a união desses componentes de maneira organizada, foi possível atingir o resultado desejado para o sistema controlador da operação e dos robôs.

2 Implementação

2.1 Configuração

Durante a implementação e para a realização de testes, uma máquina com as seguintes configurações de software e hardware foi utilizada:

Sistema Operacional	Manjaro 20.2.1 Nibia
Kernel	x86-64 Linux 5.4.89-1-MANJARO
CPU	Intel Core i5-7200U @ 4x 3.1GHz
RAM	7838 MiB
GPU	GeForce 940MX
Compilador	g++ 10.2.0
Linguagem	C++
C++ Standard	14

2.2 Estruturas Utilizadas

Os principais aspectos desse software são o gerenciamento das ordens por parte dos robôs e a comunicão da base com os robôs, seja para transmitir ordens, receber os recursos ou outra ação.

Tratando do gerenciamento das ordens por parte dos robôs, foi escolhida a estrutura de fila para as ordens de comando, enquanto as ordens diretas não são armazenadas. A escolha da estrutura se deu devido a sua política de *primeiro a entrar, primeiro a sair*, que atende ao que foi descrito nos requisitos. Quanto a implementação da mesma, foi feita de maneira dinâmica, ou seja, sem tamanho fixo, principalmente pelo fato de que a quantidade de comandos que serão armazenados por cada robô é desconhecida.

Ainda sobre as ordens, inicialmente houve a implementação de duas filas, uma para as ordens diretas e outra para as ordens de comando, porém considerando que as ordens diretas deveriam ser executadas imediatamente pelo robô, a existência de uma fila para armazená-las se mostrou desnecessária. Além disso, algumas ordens de comando poderiam vir com o caractere * antecedendo-as, caracterizando uma ordem de comando com prioridade sobre as outras, para isso existe uma segunda fila responsável apenas por armazenar as ordens com prioridade. Considerando o aspecto dinâmico da fila, o custo espacial de ter uma segunda fila é pouco significativo e permite uma implementação mais clara da lógica do programa.

Outra estrutura que se mostrou necessária foi a lista encadeada, utilizada para manter o histórico de cada um dos robôs. Novamente, considerando que o número de comandos é desconhecido, não podemos assumir um tamanho fixo para o histórico, por isso a lista também foi implementada com um tamanho dinâmico. Embora a estrutura interna da lista seja muito similar à da fila, optei por implementar a lista para que a inteface da fila não fique poluída, permitindo uma melhor separação dos procedimentos que seriam necessários nas diferentes partes da aplicação, por exemplo, as listas tem um procedimento próprio para serem impressas, enquanto as filas não.

Além dos tipos nativos, as estruturas *string*, *string stream* e *file stream* foram utilizadas para leitura do arquivo de entrada e armazenamento de algumas das informações lidas. Por fim, foram criadas diversas classes com procedimentos próprios, que serão melhor explicados na próxima subseção, porém essas foram feitas sobre as estruturas aqui mencionadas.

2.3 Descrição e Organização do Código

De maneira geral, o programa foi organizado de forma a criar abstrações para os principais componentes do problema, sendo eles a base, os robôs e as ordens. O arquivo *Main.cc*, que contém o ponto de partida do programa, contém a instanciação de um objeto da classe Base, que recebe o nome do arquivo do mapa como argumento e é responsável por instanciar todos os robôs. Depois disso, dado o arquivo contendo os comandos, eles são lidos e passados para a base através do método *sendOrder*, que realiza os devidos procedimentos, considerando se são ordens diretas ou de comando, e se comunica com a interface dos robôs. Por fim, a base emite um relatório final, contendo o total de recursos coletados e aliens eliminados pelo conjunto de robôs e o programa encerra-se.

Ainda tratando da classe *Base*, de maneira sucinta, ela é responsável por se comunicar com todos os robôs, armazenados em um array simples, manter a contagem de aliens eliminados e recursos coletados, e manter um ponteiro para o mapa do planeta, que é uma instância de *PlanetMap* criada com o argumento passado anteriormente no construtor da *Base*. Os principais métodos aqui são *sendOrder* e *relatorioFinal*, responsáveis por receber os comandos e por emitir um relatório com as informações gerais após a execução de todas as operações, respectivamente.

Como foi mencionado anteriormente, a classe *Base* também é responsável por instanciar todos os 50 robôs, isso acontece no construtor da mesma, e cada um dos objetos da classe *Robot* recebe o ponteiro para o mapa do planeta, evitando assim que exista uma dessincronização da informação no mapa entre múltiplos robôs em operação, e um número de identificação. A classe *Robots* é a que possui a interface mais complexa, sendo que cada robô é caracterizado, principalmente, por possuir: duas filas para armazenamento das ordems de comando e ordens com prioridade, uma lista que mantém o seu histórico, dois inteiros com sua posição, métodos para receber ou executar ordens, imprimir seu histórico e outros métodos para checar seu estado.

A abstração para lidar com as ordens, então, consiste na em passar todos os comandos para a base, que executa diretamente nos robôs os comandos diretos, ou chama o método *receiveOrder* dos mesmos para passar as respectivas ordens de comando. Nesse método, o robô verifica se a ordem tem maior prioridade, ou seja, se o primeiro caractere é um "*", e a insere na fila adequada.

Da perspectiva do usuário, percebe-se que, dada uma instância da classe *Base*, depois de ler um comando e o robô ao qual o comando se refere, somente é necessário chamar o método *sendOrder*, enquanto a interface abstrai todo o resto do processo.

Por fim, é valido mencionar a existência da classe *PlanetMap* e da classe abstrata *Ordem*. A primeira é uma interface para melhor lidar com o mapa lido do arquivo,

oferecendo métodos para verificar a validade de posições, acessar o que está em determinada posição e também permitindo alterar o valor em uma posição de maneira segura, isto é, apenas trocando por um ".", que representa uma posição vazia, e nunca por algo diferente do que é esperado existir no mapa. A segunda é a interface usada para implementar as classes *CommandOrder* e *DirectOrder*, embora essa última não tenha sido usada, ambas podem funcionar como nós na fila de ordens de cada robô, mantendo, mais notavelmente, um ponteiro para o próximo nó, uma string com a ordem e um inteiro que guarda o robô ao qual a ordem pertence.

3 Instruções de 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 os arquivos contendo o mapa e os comandos estão no mesmo diretório que ele:

```
./meu_executavel nomeDoArquivoDoMapa.txt nomeDoArquivoComComandos.txt
```

O executável gerado pelo Makefile tem o nome padrão *run.out*, então, exemplificando novamente, assumindo que os arquivos *mapa.txt* e *comandos.txt* estão no mesmo diretório que o executável, basta executar, sendo que o primeiro argumento é o nome do arquivo contendo o mapa e o segundo é o nome do arquivo contendo os comandos:

```
./run.out mapa.txt comandos.txt
```

3.1 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
```

3.2 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.

3.3 Testes

Durante o desenvolvimento, uma suíte com testes para vários dos métodos das classes foi sendo desenvolvida, visando tornar o processo de desenvolvimento mais robusto e permitir modificações no código com mais segurança. Para executar a suíte de testes, o processo é o seguinte:

1. Entre na pasta *Projeto*
2. Entre na pasta *src* e abra o arquivo *Main.cc*
3. Com o arquivo aberto, comente todas as linhas dele
4. Agora, volte para a raiz da pasta *Projeto*
5. Execute o comando *make test* e o programa será testado

É importante notar que é vital comentar a função *main* no arquivo *Main.cc*, pois caso isso não seja feito, no momento de compilar o arquivo de testes o compilador verá que existem dois possíveis pontos de partida para o código e não concluirá o processo.

4 Análise de Complexidade

Nessa seção será feita a análise do custo espacial e temporal dos principais métodos que podem ser executados e de cada um dos possíveis comandos que o robô pode receber.

4.1 Ordens de Comando

- **Mover, Coletar, Eliminar**
  Essas 3 diferentes ordens de comando, considerando unicamente os seus respectivos procedimentos, tem custo espacial e temporal constante, já que todas consistem de procedimentos que não dependem do tamanho da entrada, como checar se uma certa posição na matriz do mapa está ocupada por algo e mover um robô para lá, ou executar uma operação de inserção em uma lista encadeada.
  Complexidade assintótica *temporal:* $\Theta(1)$
  Complexidade assintótica *espacial:* $\Theta(1)$

4.2 Ordens Diretas

- **Ativar, Retornar**
  Essas duas ordens diretas tem complexidade espacial e temporal constante, considerando unicamente a execução das mesmas. Pois os procedimentos executados não variam conforme o tamanho da entrada. Por exemplo, na ativação do robô, uma condição é checada e então o valor da variável *activated* do robô é alterado e informações são impressas na saída, enquanto no retorno algumas checagens são realizadas, os recursos do robô transferidos para a base e informações são impressas na saída.
  Complexidade assintótica *temporal:* $\Theta(1)$
  Complexidade assintótica *espacial:* $\Theta(1)$
- **Executar**
  A complexidade espacial na execução de uma ordem **EXECUTAR** é linear, ou seja, cresce linearmente junto com o crescimento da entrada(ordens). Tendo em vista que o método utiliza as duas filas do robô, e as filas crescem linearmente.
  Tratando da complexidade temporal, a complexidade também é linear, considerando que o método percorre as duas filas por completo.
  Complexidade assintótica *temporal:* $O(n)$
  Complexidade assintótica *espacial:* $O(n)$
- **Relatorio**
  Para execução do comando relatório, é necessário percorrer e imprimir toda a lista contendo o histórico, portanto, evidentemente, a complexidade temporal é linear, assim como a espacial, já que a lista cresce linearmente conforme a entrada.
  Complexidade assintótica *temporal:* $O(n)$
  Complexidade assintótica *espacial:* $O(n)$

4.3 Outras Operações

Vamos tratar aqui das operações mais gerais, que controlam o fluxo do programa.

- **Limpar histórico**
  A cada vez que é necessário limpar o histórico, considerando que ele funciona sobre a estrutura de uma lista encadeada, é necessário passar por cada um dos nós e apagá-los, portanto é uma operação de custo temporal e espacial linear, já que a lista cresce linearmente com o aumento dos comandos na entrada..
- **Entrada e sendOrder**
  Um dos métodos chamados mais vezes é o *sendOrder*, da base, que é usado para repassar as ordens para os robôs ou solicitar a execução direta das mesmas. Esse método tem custo temporal linear, tendo em vista que caso um comando **EXECUTAR** seja recebido, ele pode ter que percorrer as duas listas, por exemplo. Embora o seu caso médio seja constante, já que só precisa checar as condições e repassar o comando para um robô.
  Esse método é chamado γ vezes durante a leitura do arquivo de comandos, onde γ é o número de comandos lidos do arquivo.

5 Conclusão

Diante de tudo que foi exposto, é possível concluir que o gerenciamento dos robôs e coordenação das operações de extração do Composto Z poderá ser realizado com sucesso. Através de um programa que organiza os robôs com a base e repassa as ordens de comando para os mesmos, que as guardam em uma fila e esperam o momento que recebem a ordem de execução para fazer o que está armazenado. Depois de concluídas todas as operações, os robôs podem retornar a base, transferir recursos, contribuir para a contagem de aliens eliminados e exibir seu histórico. Ainda, a base emite um relatório final com todos os recursos coletados e a quantidade de aliens eliminados depois de uma execução com sucesso de um arquivo de comandos.

Através do uso das filas, com política de *first in, first out*, foi possível implementar de maneira muito transparente o armazenamento das ordens de comando em cada um dos robôs. A utilização de uma implementação da fila encadeada se mostrou mais favorável, em detrimento do arranjo que tem tamanho fixo, já que a quantidade de ordens é desconhecida e em uma operação mais complexa o limite de comandos armazenados, escolhido arbitrariamente, poderia ser uma fator muito limitante.

Mostrou-se essencial o planejamento prévio e dedicação anterior ao ínicio da escrita do código, já que as diversas relações entre diferentes entidades no programa precisavam ser bem elaboradas para evitar retrabalho ou grandes modificações durante a escrita do programa. Como aluno da disciplina de Estrutura de Dados[1], e aplicando conhecimentos de disciplinas anteriores, os conceitos relacionados às especificidades de cada uma das estruturas, análise de complexidade e boas práticas de desenvolvimento se mostraram essenciais para finalizar o projeto com sucesso e em tempo hábil.

Por fim, através da implementação de uma solução para os problemas descritos, foi possível verificar a importância de conhecer técnicas para modelagem do processo de desenvolvimento de software, destacando-se a dificuldade de escolher estruturas adequadas e a forma de representar as relações entre os atores através da linguagem de programação C++.

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.