Documentação da Extração-Z

Augusto Carvalho Porto Pereira

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brazil

1.Introdução

Esta documentação trabalha com o problema de uma base extraterrestre responsável por extrair recursos essenciais para a vida na Terra. Para isso, a base conta com robôs semi-autômatos capazes de extrair os recursos supramencionados e eliminar formas de vidas alienígenas hostis, além de um mapa apontando locais onde existem recursos, inimigos ou posições de difícil acesso para os robôs que devem ser evitadas.

Para resolver o problema acima, foi implementado um sistema que verifica o mapa da região próxima à base e manipula uma fila de comandos para cada um dos 50 robôs disponíveis no posto avançado para que a base consiga enviar comandos a serem obedecidos por cada um dos robôs e garanta que eles possam emitir relatórios de sua expedição e retornar à base com segurança.

Na seção 2 desta documentação será detalhada com mais detalhes a organização do sistema e a implementação dos principais métodos, além de detalhes do hardware onde a aplicação foi criada e testada. Na seção 3 existe um breve passo-a-passo para instruir o usuário a executar o sistema. Já na quarta seção desta documentação está descrita a complexidade de espaço e de tempo para cada função do sistema, desenvolvendo por fim a complexidade geral do sistema. Por fim, na quinta e última seção, são citados os maiores aprendizados durante a realização do trabalho e quais foram as principais dificuldades enfrentadas.

2. Implementação

Para a solução do problema apresentado foi desenvolvido um sistema utilizando programação orientada a objetos subdividido entre classes que representam alguma entidade envolvida na situação. Foi criada uma classe para representar cada um dos elementos a seguir: *Base*, *Robo*, *Comando*, *CelulaComando* e *FilaEncadeadaComandos*.

A classe *Base* possui um vetor da classe *Robo* com 50 posições fixas, uma matriz de tipo char que identifica o mapa cuja base está presente na origem (posição (0,0)) e contadores de alienígenas eliminados e recursos coletados. Entre suas responsabilidades, essa classe possui os métodos *Ativar*, *Executar*, *Relatorio* e *Retornar*, que recebem como parâmetro um código inteiro representando o identificador do robô e enviam cada uma dessas ordens aos seus robôs.

Já a classe *Robo* possui o identificador, a sua coordenada atual no mapa, o número de recursos coletados e aliens eliminados, o histórico de comandos a ser exibido em seu relatório e, por fim, sua fila de comandos a serem realizados. A classe possui o método AdicionarOrdemComando que insere um comando na fila que aguardará um comando da base para executá-los, utilizando os métodos *Mover*, *Eliminar*, *Coletar* e *Limpar*.

A classe CelulaComando é uma classe auxiliar para a fila de comandos que possui um objeto *Comando* como item e um apontador para o comando seguinte na fila. Já *Comando* possui um campo tipoComando representado por um Enum que contém todos os comandos previstos pelo sistema.

Por fim, a *FilaEncadeadaComandos* representa, como o próprio nome sugere, uma fila encadeada contendo cada uma das células de comandos recebidos pelos robôs, utilizando a lógica FIFO (First in, First out). Para implementar essa fila encadeada, a classe possui uma célula cabeça apontando para o começo da fila e um ponteiro indicando o final da fila, além de uma variável tamanho para facilitar o cálculo do número de itens nela. Os seus métodos mais importantes são *InsereComando*, que insere no final da fila, *InsereComandoPrioritario*, que insere no início da fila e *RemoveComando*, que retorna o primeiro item da fila.

Utilizando todas essas classes, a função principal *(main)* recebe como parâmetro os dois arquivos de entrada que informam o mapa e a lista de comandos e, respectivamente, leem as coordenadas para preencher a matriz mapa utilizada

pela Base e lê os comandos do arquivo para inseri-los na fila de comandos de cada robô (com ou sem prioridade) ou executar as ordens diretas da base.

O sistema foi testado utilizando um Windows Subsystem for Linux (WSL) que instancia a versão 18.04 do Ubuntu. O código foi totalmente implementado em C++ (além, é claro, do arquivo Makefile) utilizando o GCC C++ Compiler (g++). A execução dos testes foi realizada em uma máquina com 8GB de memória RAM e um processador i3 de 3.30Ghz.

3. Instruções de compilação e execução

Antes de executar o programa, será necessário criar dois arquivos de tipo texto (.txt) contendo os dados do mapa e os comandos a serem realizados, conforme especificado nas especificações do trabalho. Note que o nome e a pasta desses arquivos podem ser dados a gosto do usuário, desde que ele seja escrito corretamente ao executar o comando de inicialização da aplicação que será descrito a seguir. É recomendado que esses arquivos sejam criados na pasta “TP1”.

Após criar os arquivos de leitura, o usuário deverá acessar a pasta “TP1” pelo terminal e executar o comando “bin/run.out [nomeMapa].txt [nomeComandos].txt”. Caso haja algum problema com o arquivo run.out devido ao tráfego dos arquivos via moodle, é possível executar o comando “make” para gerar novamente os arquivos “.o” e o arquivo “run.out”.

Ao executar o comando descrito acima, será impresso na saída padrão (‘stdout’) os relatórios da execução com as informações requisitadas na descrição do trabalho. Nenhum dos arquivos de entrada será alterado durante a execução do programa.

4. Análise de complexidade

Nesta seção foi registrada a análise de complexidade de espaço e tempo para cada uma das funções das classes do sistema. Ao final, será feita uma conclusão da análise de complexidade geral do código.

Para auxiliar a exibir a complexidade assintótica dos métodos abaixo, utilizaremos as variáveis: **m** para o número de linhas do mapa; **n** para o número de colunas do mapa; **c** para o número de ordens de comandos recebidas pelo robô; **o** para o número de ordens de comando totais lidos pelo programa; **t** para o total de comandos lidos no arquivo de comandos**.**

- **Base::Base**(char **mapa, int tamanhoMapaX, int tamanhoMapaY):
    - Complexidade de tempo: Mesmo que exista um laço for responsável por inicializar 50 robôs, o tempo necessário para a execução do método se mantém constante, independente das entradas, por isso a complexidade é **Θ(1)**.
    - Complexidade de espaço: Seguindo a mesma linha de raciocínio da complexidade de tempo, o espaço alocado para os 50 robôs é constante, então o espaço necessário também é **Θ(1)**.
- **Base::~Base**():
    - Complexidade de tempo: Para liberar todo o espaço da matriz mapa foi criado um laço for que é executado *m* vezes. A complexidade é **Θ(m)**.
    - Complexidade de espaço: Não é reservado para nenhuma variável no destrutor além dos auxiliares para os vetores, portanto a complexidade é **Θ(1)**.
- **Base::Ativar**(int k):
    - Complexidade de tempo: Sendo possível ou não ativar o robô, o tempo necessário é constante e, logo, **Θ(1).**
    - Complexidade de espaço: Da mesma forma, o espaço gasto é também **Θ(1).**
- **Base::Executar**(int k):
    - Complexidade de tempo: O método possui um laço while que é executado para cada comando na fila de comandos do robô. Além disso, todos os métodos chamados por essa função são somente Θ(1), portanto, a sua complexidade de tempo é **Θ(c)**.
    - Complexidade de espaço: O método não aloca espaço para além das variáveis utilizadas dentro de seu escopo. Como a complexidade de espaço das funções acessadas em seu interior são Θ(1), sua complexidade também é **Θ(1)**.
- **Base::Relatorio**(int k):

    - Complexidade de tempo: O método somente imprime dados na tela, portando é **Θ(1)**.
    - Complexidade de espaço: O método somente usa a variável identificadora do robô, portanto é **Θ(1)**.
- **Base::Retornar**(int k):
    - Complexidade de tempo: O tempo gasto na sua execução independe das entradas, portando sua complexidade é **Θ(1)**.
    - Complexidade de espaço: O espaço gasto também independe das entradas, portanto é também **Θ(1)**.

- **Robo::Robo**() e **Robo::Robo**(int id):
    - Complexidade de tempo: O tempo gasto pelos dois métodos para ambas assinaturas é igual e **Θ(1)**.
    - Complexidade de espaço: A complexidade de espaço também é semelhante e **Θ(1)**.
- **Robo::~Robo**():
    - Complexidade de tempo: O destrutor da classe Robo deleta o espaço reservado para a fila de comandos, portando tem a mesma complexidade do destrutor da FilaEncadeadaComandos. Como será descrito abaixo, sua complexidade é **Θ(c)**.
    - Complexidade de espaço: A complexidade da única função chamada por esse método é Θ(1), portanto sua complexidade é também **Θ(1)**.
- **Robo::AdicionarOrdemComando**(bool prioridade, Comando comando):
    - Complexidade de tempo: Para os dois “caminhos”, com ou sem prioridade, o tempo gasto é constante, por tanto, **Θ(1)**.
    - Complexidade de espaço: O espaço reservado para o método e as funções chamadas também é constante, portanto **Θ(1)**.
- **Robo::Mover**(char **mapa, int x, int y):
    - Complexidade de tempo: Para qualquer entrada, o tempo é constante, portanto **Θ(1)**.
    - Complexidade de espaço: Também para qualquer entrada o espaço alocado é **Θ(1)**.
- **Robo::Eliminar**(char **mapa):
    - Complexidade de tempo: Para qualquer entrada, o tempo é constante, portanto **Θ(1)**.
    - Complexidade de espaço: Também para qualquer entrada o espaço alocado é **Θ(1)**.
- **Robo::Coletar**(char **mapa):
    - Complexidade de tempo: Para qualquer entrada, o tempo é constante, portanto **Θ(1)**.
    - Complexidade de espaço: Também para qualquer entrada o espaço alocado é **Θ(1)**.
- **Robo::Limpar**():

    - Complexidade de tempo: O método é responsável somente por alterar o valor das variáveis para o valor default, portando é **Θ(1)**.
    - Complexidade de espaço: O espaço necessário para executar esse método é **Θ(1)**.

- **CelulaComando::CelulaComando**():
    - Complexidade de tempo: Esse método cria um novo Comando e, portanto, possui a mesma complexidade de tempo que o construtor de Comando. Como veremos abaixo, ele também é **Θ(1)**.
    - Complexidade de espaço: O método também obedece a complexidade de espaço do construtor de Comando, que é constante e, logo, **Θ(1)**.

- **Comando::Comando**(TipoComando tipoComando) e **Comando::Comando**(TipoComando tipoComando, int x, int y):
    - Complexidade de tempo: Para ambas implementações, o tempo gasto é o mesmo e, em ambas, **Θ(1)**.
    - Complexidade de espaço: Da mesma forma, o espaço alocado para os métodos similares é **Θ(1)**.

- **FilaEncadeadaComandos::FilaEncadeadaComandos**():
    - Complexidade de tempo: O construtor dessa classe requer tempo constante, portanto **Θ(1)**.
    - Complexidade de espaço: O construtor dessa classe também requer espaço constante, portanto **Θ(1)**.
- **FilaEncadeadaComandos::~FilaEncadeadaComandos**():
    - Complexidade de tempo: O destrutor da classe invoca um método privado que executa um laço while executado para cada comando na fila, portanto, sua complexidade é **Θ(c).**
    - Complexidade de espaço: Não é necessário alocar novo espaço para cada comando, sendo sua complexidade de espaço somente **Θ(1)**.
- **FilaEncadeadaComandos::InsereComando**(Comando comando):
    - Complexidade de tempo: Para enfileirar um novo elemento em uma fila encadeada, a complexidade de tempo é **Θ(1)**.
    - Complexidade de espaço: A complexidade de espaço para o enfileiramento também é **Θ(1)**.
- **FilaEncadeadaComandos::InsereComandoPrioritario**(Comando comando):
    - Complexidade de tempo: Para “empilhar” um elemento nessa lista encadeada, mesmo que ela siga a lógica de uma fila, a complexidade de tempo é **Θ(1)**.
    - Complexidade de espaço: A complexidade de espaço também segue a mesma ideia de um “empilhamento”, portanto é **Θ(1)**.
- **FilaEncadeadaComandos::RemoveComando**():

- Complexidade de tempo: Para desenfileirar um elemento da fila, a complexidade de tempo necessária é **Θ(1)**.
- Complexidade de espaço: A complexidade de espaço necessária também é **Θ(1)**.

Por fim, sendo verificada toda a complexidade dos métodos apresentados acima, podemos analisar a complexidade de tempo e espaço do sistema como um todo. Como toda a execução da aplicação é norteada pela execução da rotina *main*, vamos analisar a complexidade do programa a partir da sua complexidade.

Antes de avaliarmos o caso de sucesso, onde todos os arquivos serão lidos e os comandos executados, é importante destacar que caso os arquivos de entrada não sejam especificados ou o arquivo do mapa não seja encontrado a complexidade de tempo e espaço do método é Θ(1).

Considerando o cenário de execução ideal, a complexidade do programa será norteada por três blocos, referentes à leitura e ação dos dados do mapa, leitura e ação dos comandos e, por fim, a execução do destrutor nativo do compilador. Para o bloco do mapa, é alocado espaço para a matriz de tamanho m*n, portanto sua complexidade de espaço é Θ(m*n). Sobre o tempo, além do laço executado n vezes para alocar espaço da matriz do mapa, são executados dois laços aninhados que executam, cada um, m e n vezes respectivamente. Como a função de tempo desse bloco seria f(n) = mn + n + (constante), podemos dizer que essa função é Θ(mn).

Já o bloco responsável por ler os comandos é executado em um loop de t execuções onde o único método que possui complexidade de tempo diferente de constante é o método de executar (Θ(c)). Portanto, sua complexidade de tempo no pior cenário está relacionada ao número de ordens de comando para todos os robôs e o número de comandos totais, sendo assim Θ(ot). Já sua complexidade de espaço será regida pelo número de ordens de comando que serão alocadas na fila de cada robô, portanto é Θ(o).

O último bloco a ser calculado, a ser executado pelo destrutor padrão do compilador, realiza a liberação da memória alocada pelas variáveis criadas que ainda não tiveram seu espaço desalocado. Esse processo do programa tem sua complexidade de espaço e tempo regida pelo destrutor da classe *Base* que possui complexidade de tempo Θ(m) e complexidade de espaço Θ(1).

Por fim, somando a complexidade desses três blocos, podemos afirmar que a complexidade de tempo do programa desenvolvido é **Θ(mn + ot)** e sua complexidade de espaço é **Θ(mn + o)**.

5. Conclusão

Esse trabalho lidou com o problema de gerenciar uma fila de comandos enviados por uma base para cada um de seus robôs utilizando a estrutura de dados fila, mostrando a importância de que estruturas de dados e suas manipulações sejam bem aprendidos por programadores que devem saber gerenciar a ordem de leitura e escrita de informações.

Além de reforçar o conhecimento sobre estruturas de dados e tipos abstratos de dados, este trabalho ajudou a fixar o conhecimento de análise de complexidade, visto que foi necessário levantar a complexidade de cada um dos vários métodos escritos no código, gerando uma possibilidade de verificar quais seriam os “gargalos” desse código para entradas de larga escala. A principal dificuldade, porém, foi analisar essa complexidade considerando as diversas variáveis de entrada recebidas pelo sistema.

Referências

Chaimowicz, L. and Prates, R. (2020): Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.