Trabalho Prático 1
Escalonador de URLs

Vinicius Trindade Dias Abel
Matrícula: 2020007112

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

viniciustda@ufmg.br

1. Introdução

O problema proposto foi desenvolver um escalonador de URLs que adota a estratégia *depth-first* (busca em profundidade), priorizando hosts e URLs encontradas primeiro. Desta forma, o escalonador possui internamente uma fila de hosts conhecidos e cada host, por sua vez, possui uma lista de URLs conhecidas. Por fim, o escalonador possui funções que permite interagir com os hosts e URLs.

2. Método

2.1. Estrutura de Dados

A implementação do programa teve como base a estrutura de dados de uma fila encadeada, onde cada elemento da fila possui uma lista encadeada. Como pode ser visto na Figura 1.

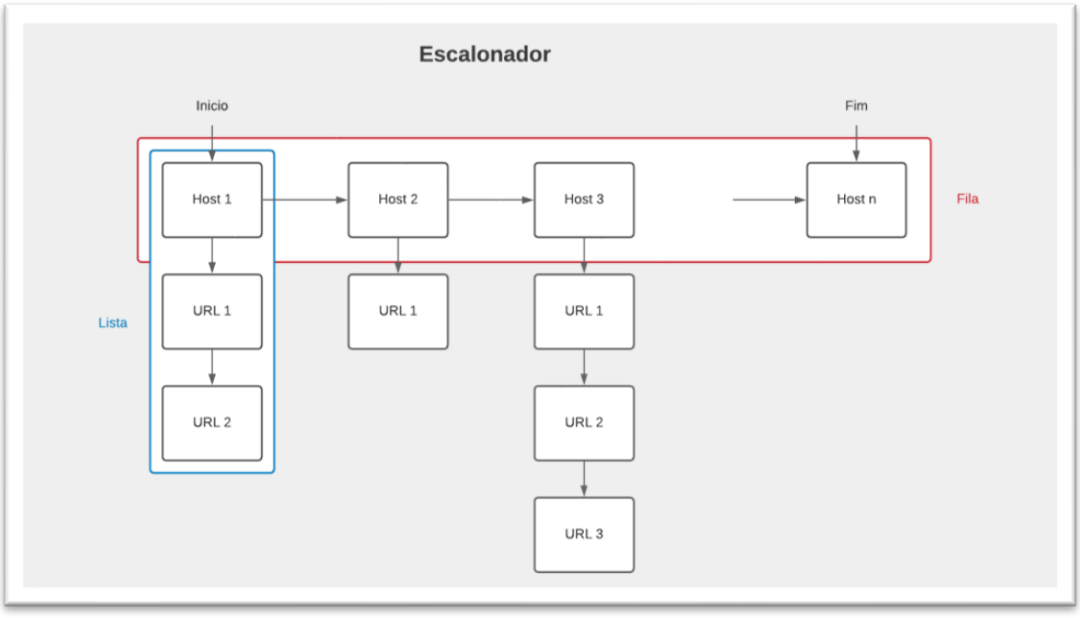

Figura 1 – Estrutura do Escalonador

Para esta implementação foram criadas classes (detalhadas no próximo tópico), sendo elas: Url, Lista e ListaEncadeada, Host, Fila e FilaEncadeada, Escalonador. Cada classe foi criada em um arquivo .hpp (e implementadas em arquivo .cpp de mesmo nome) separado das outras. Além destes, foi criado um arquivo .hpp e um .cpp de nome função_arq que possui apenas uma função, usada por mais de uma classe, que verifica se o arquivo .txt de saída está vazio.

2.2. Classes

Url

A classe Url possui uma variável num_barras do tipo int, que é o número de barras que a URL possui, sem considerar as duas barras do http://. Esse número de barras é usado para saber a profundidade da URL, que é o fator que determina a posição desta URL na lista em que está inserida.

Existe também uma variável conteudo do tipo string, que é onde fica armazenado de fato a URL.

E a última variável da classe é um ponteiro prox, que aponta para a próxima Url da lista.

Lista e ListaEncadeada

Estas classes, onde ListaEncadeada é uma classe filha de Lista, foram vistas em aula. Entretanto sofreram algumas alterações para se adequarem a solução do problema.

A primeira alteração é que todos os elementos da lista são do tipo Url.

Foi implementada uma nova função chamada ConfereUrl, que verifica se a Url passada já existe na lista. Também foram implementadas mais uma versão da função Imprime e da função Limpa, o que permitiu limitar quantos elementos são impressos ou limpos. E a função Pesquisa foi removida, porque ela tinha um papel muito semelhante ao da nova função.

Outra alteração foi que nas operações de inserção também é passado o número de barras da nova Url que será adicionada à lista.

Host

A classe Host possui uma variável conteudo do tipo string, que é onde fica armazenado de fato o host.

Também possui uma variável ponteiro prox, que aponta para o próximo Host da fila.

E possui uma ListaEncadeada de nome urls, para armazenar as URLs do host.

Fila e FilaEncadeada

Assim como as classes Lista e ListaEncadeada, Fila e FilaEncadeada foram vistas em aula e sofreram alterações se adequarem a solução do problema.

Todos os elementos da fila são do tipo Host, e foi criada a função ConfereHost que verifica se já existe o host na fila.

Escalonador

A classe Escalonador possui uma FilaEncadeada de nome hosts, para armazenar todos hosts passados para o programa.

Possui também uma variável nome_arq_saida do tipo string que armazena o nome do arquivo de saída, para facilitar quando for preciso abrir o arquivo.

E os métodos desta classe são justamente os comandos que o escalonador deve obedecer.

2.3. Main

A função main lê linha por linha do arquivo até chegar no final deste. Para cada linha, a função verifica qual o comando e chama a função do escalonador correspondente, caso seja um comando invalido, ela ignora e passa para próxima linha.

2.4. Configuração para teste

– Sistema Operacional do seu computador: Windows 11;
– Linguagem de programação implementada: C++;
– Compilador utilizado: G++ da GNU Compiler Collection;
– Dados do seu processador: i7;
– Quantidade de memória RAM: 16GB.

3. Análise de Complexidade

função ADD_URLS - complexidade de tempo: essa função é chamada n vezes. Ela possui um laço que percorre toda a url, tendo uma complexidade O(n). E ela chama as funções ConfereHost, ConfereUrl, InserePosicao e InsereInicio, sendo que InsereInicio tem complexidade O(1) e as outras tem complexidade O(n), uma vez que estas três contam com um laço de repetição de tamanho n (no caso de InserePosicao, ela não possui o laço, mas chama a função Posiciona que possui apenas um laço). Dessa forma, a complexidade assintótica de tempo dessa função é $\Theta(n^2)$, por ela ser chamada n vezes e a maior complexidade dentro dele ser O(n).

função ADD_URLS - complexidade de espaço: essa função realiza operações com a lista e com a fila, que possuem tamanho n. Assim, a complexidade assintótica de espaço dessa função é $\Theta(n^2)$, porque tem uma lista dentro de cada elemento da fila.

função ESCALONA_TUDO - complexidade de tempo: essa função possui um laço de tamanho n, e dentro deste laço ela chama as funções Imprime

e Limpa. Cada uma destas funções tem complexidade O(n), pois cada uma delas possuem apenas um laço de tamanho n. Dessa forma, a complexidade assintótica de tempo da função ESCALONA_TUDO é $\Theta(n^2)$.

função ESCALONA_TUDO - complexidade de espaço: essa função realiza operações com a lista e com a fila, que possuem tamanho n. Assim, a complexidade assintótica de espaço dessa função é $\Theta(n^2)$, porque tem uma lista dentro de cada elemento da fila.

função ESCALONA - complexidade de tempo: no pior caso, a complexidade é igual a da função ESCALONA_TUDO. No melhor caso, a complexidade é O(n), porque possui apenas os laços das funções Imprime e Limpa.

função ESCALONA - complexidade de espaço: mesmo caso da função ESCALONA_TUDO no pior caso. No melhor caso, não percorre a fila, então a complexidade é O(n).

função ESCALONA - complexidade de tempo: no pior caso, mesmo caso da função ESCALONA_TUDO, porque é preciso percorrer a fila também. No melhor caso, estamos lidando com o primeiro elemento da fila, então a fila não seria percorrida e a complexidade seria O(n).

função ESCALONA - complexidade de espaço: no pior caso, mesmo caso da função ESCALONA_TUDO, porque é preciso percorrer a fila também. No melhor caso, estamos lidando com o primeiro elemento da fila, então a fila não seria percorrida e a complexidade seria O(n).

função VER_HOST - complexidade de tempo: mesmo caso da função ESCALONA, mesmo VER_HOST não chamando a função Limpa.

função VER_HOST - complexidade de espaço: mesmo caso da função ESCALONA, mesmo VER_HOST não chamando a função Limpa.

função LISTA_HOSTS - complexidade de tempo: essa função percorre apenas um laço de tamanho n e chama a função ArquivoVazio, que tem complexidade O(n) porque ela percorre todas as linhas do arquivo. Então, a complexidade de LISTA_HOSTS é $\Theta(n)$.

função LISTA_HOSTS - complexidade de espaço: essa função trabalha apenas com a fila de hosts, então a complexidade assintótica de espaço dessa função é $\Theta(n)$.

função LIMPA_HOST - complexidade de tempo: mesmo caso da função ESCALONA, mesmo LIMPA_HOST não chamando a função Imprime.

função LIMPA_HOST - complexidade de espaço: mesmo caso da função ESCALONA, mesmo LIMPA_HOST não chamando a função Imprime.

função LIMPA_TUDO - complexidade de tempo: essa função percorre todas as listas e toda a fila, usando um laço, então a complexidade assintótica de tempo dessa função é $\Theta(n^2)$.

função LIMPA_TUDO - complexidade de espaço: essa função percorre todas as listas e toda a fila, usando um laço, então a complexidade assintótica de espaço dessa função é $\Theta(n^2)$.

Geral - complexidade de tempo: a quantidade de funções chamadas é definida pelo arquivo de entrada, que é lido linha por linha até o fim do arquivo. Com isso, a complexidade do programa no pior caso é $\Theta(n^3)$, porque é $n \times \Theta(n^2)$ onde $\Theta(n^2)$ é a maior complexidade entre as funções e n é o número de funções que o arquivo de entrada chama. No melhor caso a complexidade é $\Theta(n^2)$, porque é $n \times \Theta(n)$ onde $\Theta(n)$ é a menor complexidade entre as funções e n é o número de funções que o arquivo de entrada chama.

Geral - complexidade de espaço: a complexidade de espaço é $\Theta(n^2)$ no pior caso, porque é quando a fila e as listas são percorridas. E no melhor caso $\Theta(n)$, porque é quando só a fila ou só uma lista é percorrida.

4. Estratégias de Robustez

As classes foram implementadas de forma que apenas as funções são públicas, para evitar o acesso indevido aos atributos.

Quando é passado um comando invalido, o programa ignora e lê o próximo comando, assim não é preciso parar a execução. Se os próximos comandos estiverem certos, eles serão executados.

Se ocorrer algum erro na abertura de arquivo, o erro é mostrado.

Se for passado algum Host que não existe, sem ser no caso de adicionar, ele é ignorado.

5. Análise Experimental

Desempenho computacional: Está ligado a quantidade de funções chamadas pelo arquivo de entrada, quanto mais funções, maior o tempo de execução.

Acesso à memória: O programa foi implementado usando alocação dinâmica, então o programa não reserva um espaço na memória previamente. Dependendo do arquivo de entrada, a memoria utilizada altera. Se tiver mais hosts e URLs, mais memoria será usada.

6. Conclusões

Este trabalho lidou com um escalonador de URLs que adota a estratégia *depth-first* (busca em profundidade), na qual a abordagem utilizada para sua resolução foi uma fila em que cada elemento possuía uma lista.

Com a solução adotada, pode-se verificar que o programa busca executar as funções evitando parar mesmo com erro na entrada, ignorando entradas erradas.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados a alocação dinâmica, filas e listas, manipulação de strings e de arquivos passados como parâmetro, e a criação de estruturas de dados.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo a manipulação do arquivo .txt, trabalhar com lista e fila encadeadas, criar classes e a implementação do makefile.

7. Bibliografia

- std:basic_string. Disponível em: https://www.inf.pucrs.br/~flash/lapro2ec/cppreference/w/cpp/string/basic_string.html. Acesso em: 14 de dezembro de 2021.

- C++ - Operações com arquivos. Disponível em: https://moodle.ufsc.br/pluginfile.php/2377820/mod_resource/content/0/exercicios%20arquivos.pdf. Acesso em: 14 de dezembro de 2021.

- Argumentos em linha de comando. Disponível em: http://linguagemc.com.br/argumentos-em-linha-de-comando/. Acesso em: 14 de dezembro de 2021.

8. Instruções para compilação e execução

• Acesse o diretório TP

• Utilizando o terminal, compile o TP1 utilizando o comando: make

• Com esse comando, será gerado o arquivo tp1.exe no diretório bin e os arquivos *.o no diretório obj

• Acesse o diretório bin

• Utilizando o terminal, execute o arquivo tp1.exe junto com o arquivo *.txt utilizando o comando: tp1.exe <arquivo_de_entrada.txt>