Trabalho Prático: Escalonador de URLs

Gabriel Teixeira Carvalho

1. Introdução

Esta documentação lida com o problema de implementar um escalonador de URLs, módulo de uma máquina-de-busca responsável por definir a ordem na qual as URLs presentes na Internet devem ser coletadas. O objetivo principal desta tarefa é criar um programa que, dada uma série de URLs de diferentes hosts no formato

```
<protocolo>://<host><path>?<query>#<fragmento>
```

consiga definir a ordem na qual essas URLs devem ser coletadas, segundo a estratégia de coleta depth-first (busca em profundidade), em que todas as URLs de um host são coletadas antes de passar para o próximo. Além disso, outro foco importante foi a revisão de conceitos de classes, estruturas de dados, desempenho e robustez de código, sendo muito aplicados ao longo da implementação. Para representar um escalonador e resolver o problema citado, foi utilizada uma estrutura formada por uma fila de listas que coleta hosts conhecidos primeiro e, em seguida, URLs com menor profundidade.

A seção 2 desta documentação trata mais a fundo sobre como o programa foi implementado, enquanto na seção 3 é apresentada uma análise da complexidade temporal e espacial do trabalho. Já na seção 4, são apresentados aspectos de robustez e proteção contra erros de execução. A seção 5 destrincha o comportamento do programa em termos de tempo e espaço através de diversos experimentos realizados após a implementação do programa. A seção 6 trata de duas outras estratégias de coleta possíveis que também foram implementadas. Por fim, a seção 7 conclui e sumariza tudo que será falado nesta documentação, sendo seguida por referências bibliográficas utilizadas na confecção do código e por um manual de compilação e execução do programa.

2. Método

2.1. Ambiente de desenvolvimento

O programa foi implementado na linguagem C++, compilado pelo G++, compilador da GNU Compiler Collection. Além disso, foi utilizado o sistema operacional Ubuntu 20.04 em um Windows 10, através do Windows Subsystem for Linux (WSL2), com um processador Intel Core i7-6500U (2.50GHz, 4 CPUs) e 8192 MB RAM.

2.2. Organização, Arquivos e Funcionamento

O código está organizado em 4 diretórios na pasta raiz:

- `obj`: Esta pasta contém os arquivos `.cpp` da pasta `src` traduzidos para linguagem de máquina pelo compilador GCC e com a extensão `.o` (object).

- `bin`: Contém o arquivo executável `main`, que consiste nos arquivos .o da pasta `obj` compilados em um só arquivo pelo GCC.

- `include`: Contém os arquivos de declaração (contrato) das classes utilizados no programa, com seus atributos e métodos especificados dentro de arquivos da extensão `.h`, que serão incluídos e implementados nos arquivos de extensão `.cpp`:
    - `listadeurls.h`: Define a classe `ListaDeURLS` que possui células de `string` encadeadas através de ponteiros. Cada célula representa uma URL e aponta para a próxima célula na lista, formando uma lista de URLs. Essa classe possui métodos que permitem inserir, remover e acessar URLs em qualquer posição da lista e imprimi-las, sendo importantes para a implementação do escalonador.
    - `filadehosts.h`: Define a classe `FilaDeHosts` que possui células de `ListaDeURLS` encadeadas através de ponteiros. Cada célula representa um host com uma lista de URLs e aponta para a próxima célula na fila, formando uma fila de hosts. Essa classe possui métodos que permitem inserir uma lista no Final da Fila, remover uma lista do início e acessar uma lista em qualquer posição, sendo importantes para a implementação do escalonador.
    - `escalonador.h`: Define a classe Escalonador, formado por uma FilaDeHosts com diversos métodos importantes para a coleta de URLs, tais como adição de URLs, escalonamento, listagem dos hosts e limpeza da estrutura.
    - `memlog.h`: Define um TAD que gerencia o registro dos acessos à memória e o registro do desempenho, através de métodos utilizados sempre que há leitura ou escrita em memória.
    - `msgassert.h`: Define macros que conferem condições necessárias para o funcionamento correto do programa, abortando a execução ou imprimindo um aviso caso estas condições sejam descumpridas.

- `src`: Contém as implementações das classes declarados nos arquivos `.h` da pasta `include` em arquivos `.cpp`; Nestes arquivos estão programadas as regras, a lógica e a robustez contra erros que é o alicerce de todo o programa:
    - `listadeurls.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `listadeurls.h`. Métodos mais importantes:
        - `getItem`: Retorna a URL na posição especificada.
        - `insereInicio`: Insere uma URL no início da lista.
        - `insereFinal`: Insere uma URL no final da lista
        - `inserePosicao`: Insere uma URL na posição especificada da lista
        - `removeInicio`: Remove e retorna a URL no início da lista. Como as URLs são inseridas já na posição correta, chamar

esse método permite retornar a URL certa, segundo a ordem estabelecida.
    - `imprime`: Imprime todas as URLs da lista em um arquivo especificado
    - `limpa`: Desaloca o espaço utilizado pela lista, liberando espaço no heap e evitando vazamentos de memória.

- `filadehosts.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `filadehosts.h`. Métodos mais importantes:
    - `getHost`: Retorna o nome do host da célula na posição especificada.
    - `getItem`: Retorna o endereço da ListaDeURLS na posição especificada.
    - `insereFinal`: Insere uma ListaDeURLS no final da Fila, seguindo a lógica First In First Out dessas estruturas.
    - `limpa`: Desaloca o espaço utilizado pela fila, liberando espaço no heap e evitando vazamentos de memória.
- `escalonador.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `escalonador.h`. Métodos mais importantes:
    - `addUrl`: Alicerce do trabalho, com a ajuda da função `insereNaPosiçãoCorreta` e de expressões regulares (regex) para identificar URLs válidas, insere uma URL no escalonador segundo a profundidade das URLs, que consiste na quantidade de barras ('/') presentes na string..
    - `insereNaPosicaoCorreta`: Insere a URL na posição correta da lista através da comparação da quantidade de barras presentes na URL.
    - `escalonaTudo`: Imprime em um arquivo e remove da estrutura (escalona) todas as URLs presentes no escalonador.
    - `escalona`: Escalona uma quantidade especificada de URLs do escalonador segundo a ordem estabelecida.
    - `escalonaHost`: Escalona uma quantidade especificada de URLs do host especificado.
    - `verHost`: Exibe todas as URLs do host especificado na ordem de prioridade.
    - `listaHosts`: Exibe todos os hosts do escalonador segundo a ordem em que foram conhecidos.
    - `limpaHost`: Limpa a lista de URLs do host especificado, deixando o host vazio.
    - `limpaTudo`: Limpa todas as URLs e hosts do escalonador.
- `main.cpp`: Este arquivo implementa a função principal do programa, responsável por controlar o fluxo de execução do programa. Esse fluxo é seguido de acordo com um arquivo de

comandos passado como parâmetro pela linha de comando com chamadas para os métodos do escalonador.

- `memlog.c`: Neste arquivo está presente a implementação do TAD que registra os acessos de memória e desempenho de tempo. Métodos mais importantes:
  - `leMemLog`: Registra em um arquivo uma operação de leitura de um elemento realizada, com informações como tempo, endereço acessado, tamanho do elemento.
  - `escritaMemLog`: Registra em um arquivo uma operação de escrita de um elemento realizada, com informações como tempo, endereço acessado, tamanho do elemento.
  - `defineFaseMemLog`: Separa as fases de execução do programa, fundamental para a análise experimental.

Além disso, há um arquivo `makefile`, responsável pela compilação modularizada do programa, permitindo compilar somente os módulos necessários, ao invés de recompilar todos os arquivos a cada mudança.

2.3. Detalhes de implementação

As estruturas de dados fundamentais para o trabalho, utilizadas para representar o escalonador, foram filas e listas. Ambas foram implementadas com uma célula cabeça vazia, que guarda o endereço do primeiro elemento da estrutura e aponta para o próximo. A lista permite acesso, inserção e remoção de URLs em qualquer posição, enquanto a fila só permite inserção e remoção de hosts nas extremidades. Normalmente, as filas não permitem acesso de elementos aleatórios, porém, para implementar métodos em que algum host específico precisa ser acessado, foi necessário permitir acessar qualquer elemento.
Outra estrutura importante é o TAD `memlog_tipo`, que guarda informações necessárias para o registro de performance.

As operações que o usuário pode executar no programa são as seguintes:

- `ADD_URLS <quantidade>`: adiciona ao escalonador as URLs informadas nas linhas seguintes. O parâmetro `<quantidade>` indica quantas linhas serão lidas antes do próximo comando.
- `ESCALONA_TUDO`: escalona todas as URLs seguindo as regras estabelecidas previamente. Quando escalonadas, as URLs são exibidas e removidas da lista.
- `ESCALONA <quantidade>`: limita a quantidade de URLs escalonadas.
- `ESCALONA_HOST <host> <quantidade>`: são escalonadas apenas URLs deste host.
- `VER_HOST <host>`: exibe todas as URLs do host, na ordem de prioridade.
- `LISTA_HOSTS`: exibe todos os hosts, seguindo a ordem em que foram conhecidos.
- `LIMPA_HOST <host>`: limpa a lista de URLs do host.

A definição das operações a serem executadas fica a cargo do usuário, que deve passar como parâmetro para o programa, pela linha de comando, um arquivo de texto contendo as operações desejadas. O programa reconhece o nome do arquivo por meio de regex e cria um arquivo, com o mesmo nome do arquivo de entrada acrescido do sufixo '-out',

onde serão armazenados os resultados das operações desejadas. A leitura dos comandos e parâmetros dentro do arquivo de entrada é feita com a ajuda de regex, que permite reconhecer diferentes comandos e separar seus argumentos.

Por fim, cabe ressaltar que todas as passagens de matrizes por parâmetro para as funções e métodos é feita por referência, passando o endereço ao invés de copiar a matriz em questão. Isto economiza grandes quantidades de memória a cada chamada de função.

3. Análise de Complexidade

Nesta seção será analisada a complexidade tanto de tempo como de espaço das funções e métodos descritos acima. em complexidade de tempo:

Método `ListaDeURLS::getItem` – complexidade de tempo:

Esse método percorre a lista até chegar à posição especificada (N) e retorna o elemento na posição N. Portanto, a complexidade assintótica de tempo é Θ(N).

Método `ListaDeURLS::getItem` – complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Lista, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `ListaDeURLS::insereInicio` - complexidade de tempo:

Esse método insere uma célula nova no início, ajusta os atributos e ponteiros das células e o tamanho da lista. Sua complexidade é constante, Θ(1).

Método `ListaDeURLS::insereInicio` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Lista, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `ListaDeURLS::insereFinal` - complexidade de tempo:

Esse método insere uma célula nova no final, ajusta os atributos e ponteiros das células e o tamanho da lista. Sua complexidade é constante, Θ(1).

Método `ListaDeURLS::insereFinal` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Lista, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `ListaDeURLS::inserePosicao` - complexidade de tempo:

Esse método percorre a lista até chegar uma posição antes da especificada (N) para poder inserir uma célula nova, O(N). Então, ajusta os atributos e ponteiros das células e o tamanho da lista. Sua complexidade é constante, Θ(N).

Método `ListaDeURLS::inserePosicao` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Lista, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `ListaDeURLS::removeInicio` - complexidade de tempo:

Esse método remove uma célula do início, ajusta os atributos e ponteiros das células e o tamanho da lista. Sua complexidade é constante, Θ(1).

Método `ListaDeURLS::removeInicio` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Lista, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `ListaDeURLS::imprime` - complexidade de tempo:

Esse método percorre toda a lista imprimindo cada elemento. Portanto, a complexidade assintótica de tempo varia com o tamanho da lista (N), logo, é Θ(N).

Método `ListaDeURLS::imprime` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Lista, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `ListaDeURLS::limpa` - complexidade de tempo:

Esse método percorre a lista deletando cada célula. Portanto, a complexidade assintótica de tempo varia com o tamanho da lista (N), logo, é Θ(N).

Método `ListaDeURLS::limpa` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Lista, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `FilaDeHosts::getHost` - complexidade de tempo:

Esse método percorre a fila até chegar à posição especificada (N) e retorna o host da célula na posição N. Portanto, a complexidade assintótica de tempo é Θ(N).

Método `FilaDeHosts::getHost` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Fila, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `FilaDeHosts::getItem` - complexidade de tempo:

Esse método percorre a lista até chegar à posição especificada (N) e retorna o elemento na posição N. Portanto, a complexidade assintótica de tempo é Θ(N).

Método `FilaDeHosts::getItem` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Fila, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `FilaDeHosts::insereFinal` - complexidade de tempo:

Esse método insere uma célula nova no final, ajusta os atributos e ponteiros das células e o tamanho da fila. Sua complexidade é constante, Θ(1).

Método `FilaDeHosts::insereFinal` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Fila, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `FilaDeHosts::limpa` - complexidade de tempo:

Esse método percorre a fila deletando cada célula. Portanto, a complexidade assintótica de tempo varia com o tamanho da lista (N), logo, é Θ(N).

Método `FilaDeHosts::limpa` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Fila, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Função `Escalonador::insereNaPosicaoCorreta` - complexidade de tempo:

- Se o escalonador está vazio, o método `ListaDeURLs::insereInicio` é chamado (Θ(1)).
- Se o escalonador tem 1 elemento, um dos métodos `ListaDeURLs::insereInicio` ou `ListaDeURLs::insereFinal` é chamado (Θ(1)).
- Se o escalonador tem mais de 1 elemento, a Lista de tamanho N é percorrida (O(N)) e para cada iteração o método `ListaDeURLs::getItem` é chamado (O(N)), portanto, a complexidade assintótica de tempo é $\Theta(N^2)$.

- Melhor caso: Θ(1)
- Pior caso: $\Theta(N^2)$

Função `Escalonador::insereNaPosicaoCorreta` - complexidade de espaço:

Essa função requer uma quantidade constante de memória independentemente da entrada e a `ListaDeURLs` é passada como parâmetro por referência, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `Escalonador::addUrl` - complexidade de tempo:

Esse método utiliza um regex constante para todas as URLs passadas como parâmetros. Para uma URL de tamanho M, a complexidade para identificar uma URL é Θ(M).

No caso em que o host da URL não está presente no escalonador, esse método chama a `ListaDeURLS::insereInicio` (Θ(1)) seguida da `FilaDeHosts::insereFinal` (Θ(1)), logo, sua complexidade assintótica de tempo é Θ(M).

No caso em que o host da URL não está presente no escalonador, a fila do escalonador é percorrida até achar o host passado como parâmetro (Θ(P = posição do host)), e a função `insereNaPosicaoCorreta` é chamada (Θ(N^2) no pior caso). Logo, a complexidade assintótica de tempo nesse caso é Θ(P*N^2).

Melhor caso: Θ(M).

Pior caso: Θ(P*N^2).

Método `Escalonador::addUrl` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para `ListaDeURLS`, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `Escalonador::escalonaTudo` - complexidade de tempo:

Esse método percorre a fila imprimindo todos os elementos de cada lista. Supondo que o tamanho da fila seja M e o tamanho de cada lista seja N, a complexidade assintótica de tempo é Θ(M*N).

Método `Escalonador::escalonaTudo` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para `ListaDeURLS`, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `Escalonador::escalona` - complexidade de tempo:

Esse método aloca escalona a quantidade especificada de URLs (N), utilizando o método `ListaDeURLS::removeInicio` (Θ(1)) a cada escalonamento. Portanto, sua complexidade assintótica de tempo é Θ(N).

Método `Escalonador::escalona` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para `ListaDeURLS`, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `Escalonador::escalonaHost` - complexidade de tempo:

Esse método percorre a fila até achar o host especificado (Θ(N = posição do host)) e então escalona a quantidade especificada de URLs (M) com o método `ListaDeURLS::removeInicio` (Θ(1)). Logo, sua complexidade assintótica de tempo é Θ(M + N);

Método `Escalonador::escalonaHost` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para `ListaDeURLS`, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `Escalonador::verHost` - complexidade de tempo:

Esse método percorre a fila até achar o host especificado (Θ(N = posição do host)) e então imprime os elementos da lista (Θ(M = tamanho da Lista)). Logo, sua complexidade assintótica de tempo é Θ(M + N);

Método `Escalonador::verHost` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para `ListaDeURLS`, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `Escalonador::listaHosts` - complexidade de tempo:

Esse método percorre a fila imprimindo os hosts. Para cada iteração, chama o método `FilaDeHosts::GetItem` (Θ(N = posição do elemento na fila)). Logo, sua complexidade assintótica de tempo é $\Theta(1 + 2 + ... + M) = \Theta(M^2)$;

Método `Escalonador::listaHosts` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, logo, é Θ(1) em espaço.

Método `Escalonador::limpaHost` - complexidade de tempo:

Esse método percorre a fila até achar o host especificado (Θ(M = posição do host)) e então chama o método ListaDeURLS::limpa (Θ(N = tamanho da lista)). Logo, sua complexidade assintótica de tempo é Θ(M + N);

Método `Escalonador::limpaHost` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para `ListaDeURLS`, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `Escalonador::limpaTudo` - complexidade de tempo:

Chama o método `FilaDeHosts::limpa` que é (Θ(N = tamanho da fila)). Logo, sua complexidade assintótica de tempo é Θ(N).

Método `Escalonador::limpaTudo` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, é Θ(1) em espaço.

Método `memlog::leMemLog` - complexidade de tempo:

Esse método realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Método `memlog::leMemLog` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Método `memlog::escreveMemLog` - complexidade de tempo:

Esse método realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Método `memlog::escreveMemLog` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Método `memlog::defineFaseMemLog` - complexidade de tempo:

Esse método realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Método `memlog::defineFaseMemLog` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Além disso, para o registro dos acessos à memória, a cada operação de leitura ou escrita será realizada uma operação de impressão de complexidade O(1). Essa opção está desativada por padrão no código, mas pode ser ativada facilmente para que as análises sejam feitas. Isso não muda a ordem de complexidade da operação escolhida, porém, aumenta a constante que multiplica a função de complexidade da operação. Portanto, essa opção pode reduzir bastante a performance do código.

4. Estratégias de Robustez

Ao longo do código, são utilizadas as macros `avisoAssert` e `erroAssert` e impressões no terminal. As macros asseguram certas condições importantes para o programa e podem gerar uma mensagem de aviso no caso do `avisoAsser`t ou gerar uma mensagem e abortar a execução do programa no caso do `erroAssert`. As impressões no terminal são utilizadas para que o usuário saiba que algum dos comandos

caiu em um caso de exceção. Os mecanismos de programação defensiva e robustez utilizados foram:

- Asserção de nome do arquivo passado por parâmetro – aborta o programa se o nome do arquivo passado por parâmetro não for especificado.
- Asserção de abertura de arquivo – aborta o programa se o arquivo de comandos ou de saída não forem abertos.
- Impressão de aviso caso um comando inexistente seja passado para o programa.
- Impressão de aviso caso a URL tenha uma extensão inválida.
- Impressão de aviso caso a URL seja inválida.
- Impressão de aviso caso o host não esteja presente em algum método que recebe um host como parâmetro.
- Impressão de aviso caso haja uma tentativa de remoção de um elemento em uma estrutura vazia.
- Impressão de aviso caso haja uma tentativa de acesso a uma posição inválida em uma estrutura.

5. Análise Experimental

Essa seção compila uma série de experimentos de diferentes tipos que analisam a performance do programa por meio dos registros do TAD `memlog_tipo`.

5.1. Análise de tempo

Esse experimento tem como objetivo medir o tempo de execução ao performar as operações implementadas no código com diferentes cargas de trabalho

O tempo é obtido ao subtrair o tempo inicial do tempo final, os quais estão presentes no arquivo de registro de desempenho. Foram geradas cargas de trabalho que variam em três dimensões: Profundidade das URLs, Quantidade de Hosts e Quantidade de URLs. Foi feita a mesma bateria de testes para cada carga: o programa é executado 4 vezes e a média de tempo entre as execuções é calculada. Foram realizados testes com cada uma das dimensões nos valores 5, 10, 20, 40, 80, 160 e 320 enquanto as outras duas eram mantidas constantes em 5. Além disso, foi realizada uma bateria de testes com as 3 dimensões variando juntas.

Ao analisar os dados obtidos e traçando um gráfico com a ajuda do programa GNUPLOT, percebe-se que o programa tem um comportamento polinomial com complexidade

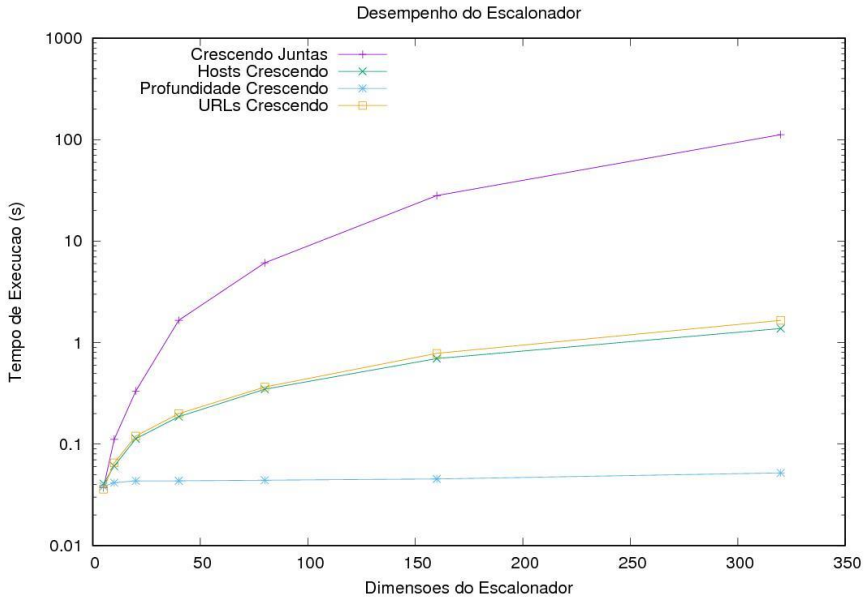

Figura 1: gráfico de desempenho de tempo

Além disso, analisando o perfil de execução do código com o programa GPROF, chega-se a outras conclusões importantes:

Do tempo de execução total do programa, mais de 90 % corresponde à execução do método `addUrl` e, dessa porcentagem, mais de 80% corresponde à identificação de URLs válidas com regex (sem o registro de acesso ativado).

```
                0.00    1.04    1601/1601        main [1]
[2]     93.9    0.00    1.04    1601         Escalonador::addUrl(std::__cxx11::basic_string<char, std::char_trai
                0.00    0.87    1601/1606        std::__cxx11::basic_regex<char, std::__cxx11::regex_traits<char
                0.01    0.15    1596/1596        insereNaPosicaoCorreta(ListaDeURLS*, std::__cxx11::basic_string
                0.00    0.01    1601/1606        bool std::regex_match<std::char_traits<char>, std::allocator<ch
                0.00    0.00    1601/1606        std::__cxx11::basic_regex<char, std::__cxx11::regex_traits<char
                0.00    0.00    6404/6411        std::__cxx11::match_results<__gnu_cxx::__normal_iterator<char c
                0.00    0.00    6404/6411        std::__cxx11::sub_match<__gnu_cxx::__normal_iterator<char const
                0.00    0.00    4877/4891        FilaDeHosts::getTamanho() [637]
                0.00    0.00    4872/4877        FilaDeHosts::getHost[abi:cxx11](int) [638]
                0.00    0.00    4872/102487      __gnu_cxx::__enable_if<std::__is_char<char>::__value, bool>::__
                0.00    0.00    3197/6664547     escreveMemLog(long, long, int) [235]
                0.00    0.00    1606/1748577     void std::__cxx11::basic_string<char, std::char_traits<char>, s
                0.00    0.00    1601/1606        std::__cxx11::match_results<__gnu_cxx::__normal_iterator<char c
                0.00    0.00    1601/1615        ListaDeURLS::ListaDeURLS() [749]
                0.00    0.00    1601/1606        std::__cxx11::match_results<__gnu_cxx::__normal_iterator<char c
                0.00    0.00    1596/1601        FilaDeHosts::getItem(int) [934]
                0.00    0.00       5/14          ListaDeURLS::insereInicio(std::__cxx11::basic_string<char, std:
                0.00    0.00       5/5           FilaDeHosts::insereFinal(ListaDeURLS, std::__cxx11::basic_strin
```

Figura 2: perfil de execução do programa

Além disso, o registro de acesso pode aumentar consideravelmente o custo para executar o código, chegando a tomar quase 10% do tempo de execução total.

```
-----------------------------------------------
                0.02    0.00 2617707/5294764     leMemLog(long, long, int) [24]
                0.02    0.00 2677057/5294764     escreveMemLog(long, long, int) [22]
[23]     8.9    0.04    0.00 5294764         clkDifMemLog(timespec, timespec, timespec*) [23]
```

Figura 3: perfil de execução do programa com o registro de acesso ativado

5.2. Localidade de referência

Outra forma de estudar a performance do programa é fazendo um Mapa de Acesso à Memória, o qual permite enxergar claramente como são feitos os acessos à memória do computador durante a execução das operações, tanto no tempo quanto no espaço.

Esse tipo de observação leva em conta os princípios de Localidade de Referência Espacial e Temporal, que dizem que um acesso ao endereço X no tempo T implica que acessos ao endereço X + dX no tempo T + dT se tornam mais prováveis à medida que dX e dT tendem a zero.

Para facilitar essa análise, as funções registradoras (`leMemLog` e `escreveMemLog`) registram uma fase e um ID em todo registro:

- Um ID identifica a qual classe o registro se refere e é atribuído à classe no momento em que são feitos os registros de acesso, pois no arquivo respectivo de cada classe, as funções registradoras estão com o ID correspondente. O ID 0 corresponde à ListaDeURLS, o ID 1 corresponde à FiladeHosts e o ID 2 corresponde ao Escalonador.
- Uma fase representa um intervalo de tempo no qual as operações realizadas são semelhantes, o que permite separar o registro em fases com diferentes análises. As fases são definidas através da função `defineFaseMemLog`, dentro de cada operação do código. Fase 0: inicialização das estruturas. Fase 1: ADD_URL. Fase 2: ESCALONA_TUDO. Fase 3: ESCALONA. Fase 4: ESCALONA_HOST. Fase 5: VER_HOST Fase 6: LISTA_HOSTS. Fase 7: LIMPA_HOST. Fase 8: LIMPA_TUDO.

Esse estudo foi feito a partir da geração de gráficos com tempo no eixo X e endereço de memória no eixo Y, após análises dos registros gerados com o registro de acesso ativo ao realizar as operações mais relevantes do escalonador: `AddUrl`, `ListaHosts`, `EscalonaTudo` e `LimpaTudo` com dimensões Profundidade = 5, URLs = 5 e Hosts = 5.

5.2.1. ListaDeURLS

Analisando a localidade de referência da Classe `ListaDeURLS`, percebe-se uma concentração de acessos em duas faixas distantes entre si. Além disso, dentro de cada faixa, os acessos também se mostram consideravelmente separados em algumas ocasiões. Por isso, para melhorar o desempenho do programa, seria válido analisar formas de melhorar a localidade de referência dessa Classe, evitando movimentações desnecessárias através da memória. Um exemplo de melhoria possível seria a separação dos métodos que acessam posições afastadas na memória em classes diferentes, evitando acessos alternados a localidades distantes. Dito isso, a localidade de referência dessa

classe tem seus pontos positivos e negativos, com acessos concentrados em faixas, mas em faixas separadas entre si.

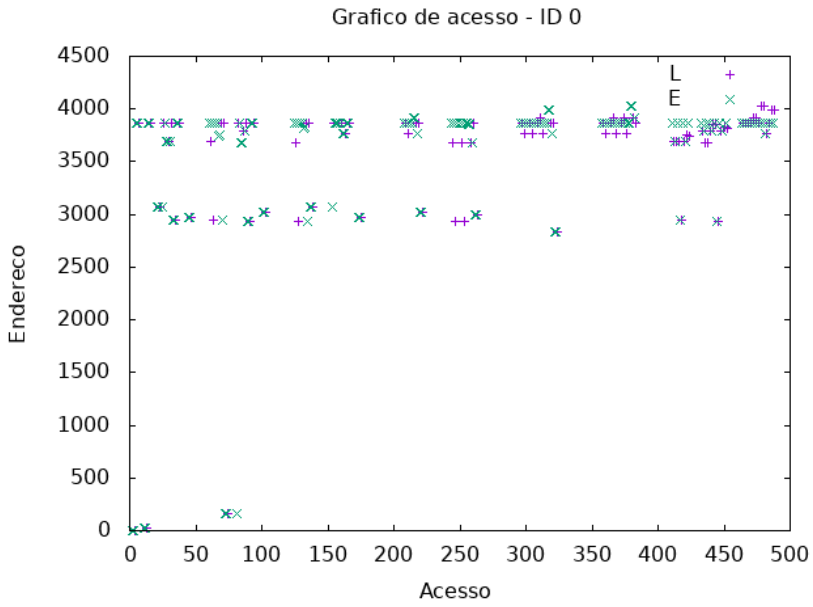

Figura 4: Mapa de acesso à memória da classe ListaDeURLS

5.2.2. FilaDeHosts

Analisando a localidade de referência da Classe `FilaDeHosts`, percebe-se a mesma dispersão presente na classe ListaDeURLS, com faixas distantes entre si. Porém, nesse caso, os acessos estão mais concentrados do que no anterior, cabendo uma análise a respeito da necessidade de uma refatoração do código para melhoria de desempenho.

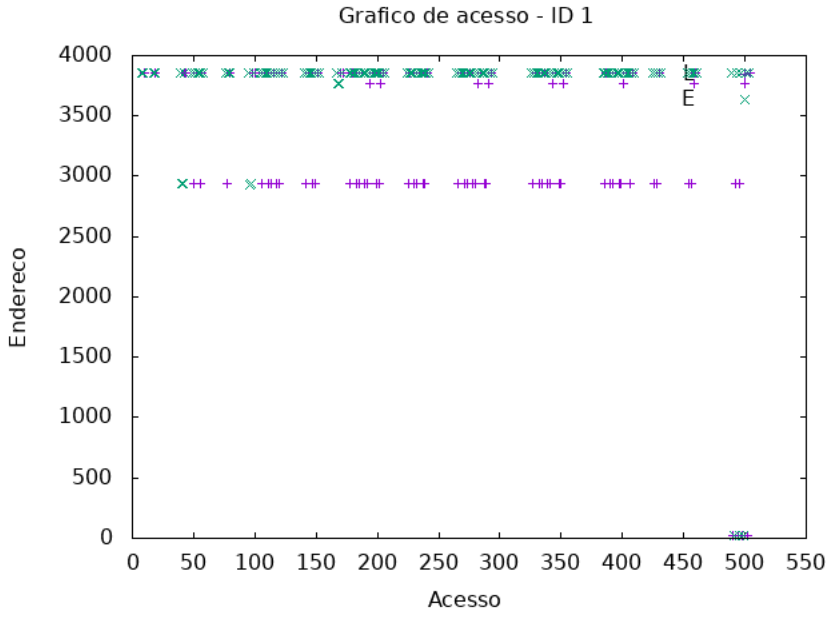

Figura 5: Mapa de acesso à memória da classe FilaDeHosts

5.2.3. Escalonador

No caso do escalonador, percebe-se acessos concentrados na mesma faixa de endereço. Indicando uma boa localidade de referência. Como a maior parte das operações realizadas pelo `Escalonador` são feitas a partir da chamada de métodos das outras duas classes, o número de acessos é bem menor nessa classe do que nas outras.

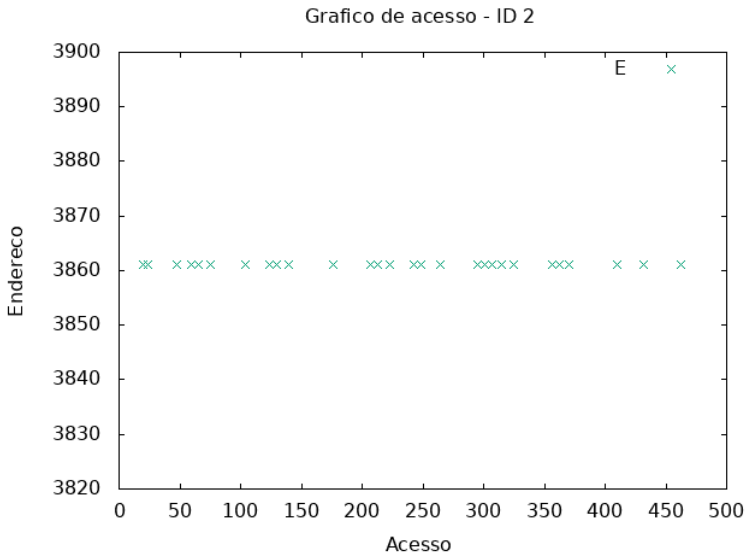

Figura 6: Mapa de acesso à memória da classe Escalonador

5.3. Distância de pilha

Outra maneira de analisar a performance do programa e que corrobora com o que já foi analisado anteriormente é a distância de pilha (DP). Cada vez que um endereço é acessado, ele é colocado no topo de uma pilha inicialmente vazia. Quando o endereço é acessado novamente, sua posição na pilha é a distância de pilha.
Fazendo uma soma ponderada de todas as distâncias de pilha com suas frequências de um conjunto de procedimentos, chega-se à distância de pilha desse conjunto e quanto maior ele seja, pior é a localidade de referência da operação. Isso se dá, pois, um mesmo endereço demora mais a ser acessado novamente se sua distância de pilha é alta, ferindo o princípio da localidade de referência. Analisando esses aspectos, é possível tirar algumas conclusões a respeito das diferentes operações e classes.

No caso da operação addUrl, percebe-se aquilo que já foi constatado nos mapas de acesso: A classe ListaDeURLS não acessa a memória tão bem, pois possui uma distância de pilha considerável e com distâncias espalhadas. A classe FilaDeHosts possui comportamento semelhante, mas com distâncias mais concentradas, seguindo melhor o princípio da localidade de referência. O mesmo vale para a operação escalonaTudo, mas com escala menor pois menos operações são realizadas nela.
No caso das operações listaHosts e limpaTudo, todo o trabalho é realizado pela classe FilaDeHosts, demonstrando mais uma vez uma boa distância de pilha.

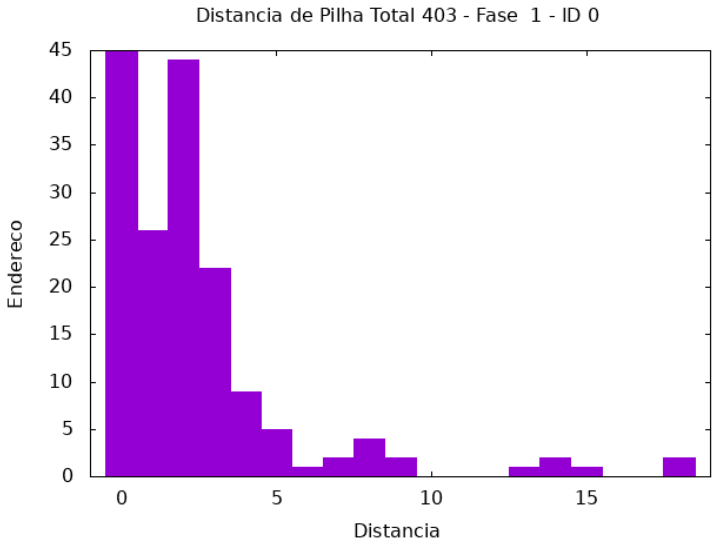

Figura 7: Histograma de distância de pilha da classe ListaDeURLs na operação addUrl

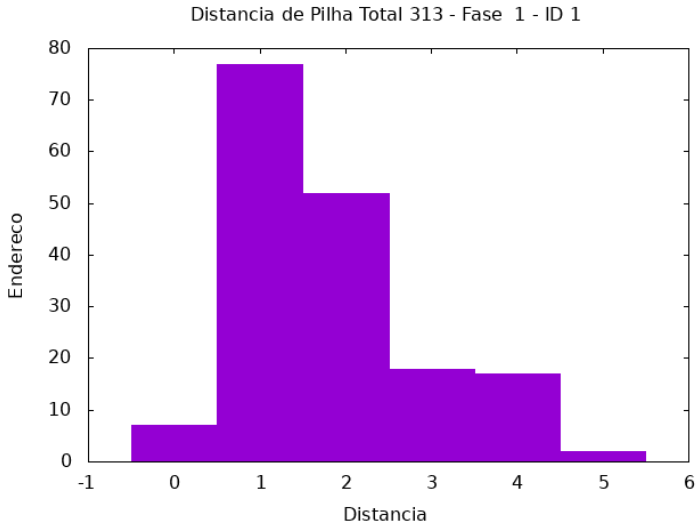

Figura 8: Histograma de distância de pilha da classe ListaDeURLs na operação addUrl

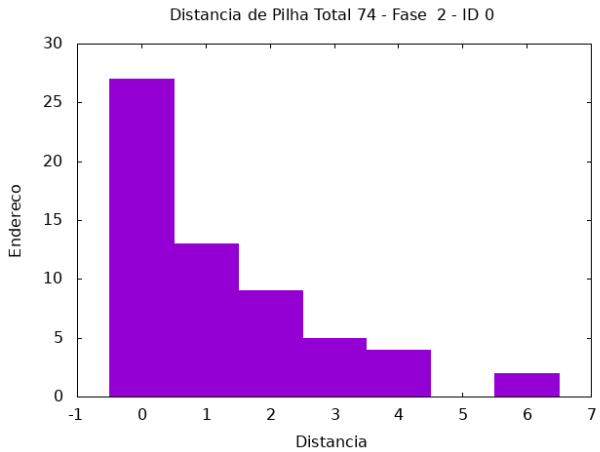

Figura 9: Histograma de distância de pilha da classe ListaDeURLs na operação escalonaTudo

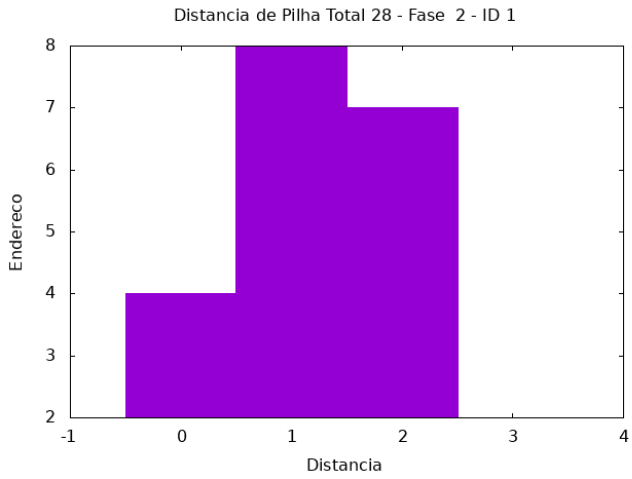

Figura 10: Histograma de distância de pilha da classe FilaDeHosts na operação escalonaTudo

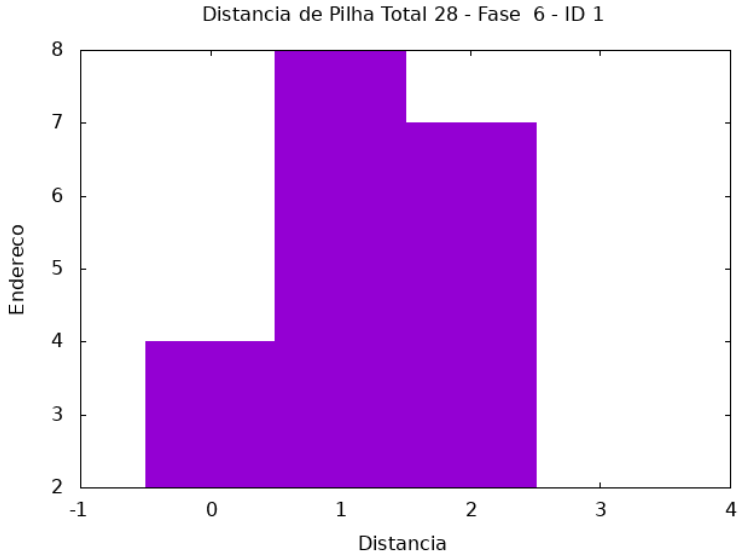

Figura 11: Histograma de distância de pilha da classe FilaDeHosts na operação listaHosts

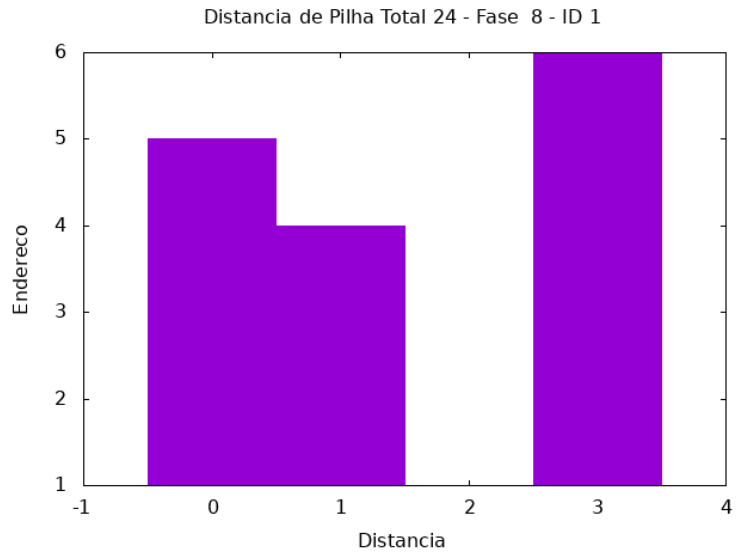

Figura 12: Histograma de distância de pilha da classe filaDeHosts na operação limpaTudo

6. Conclusão

Este trabalho lidou com a tarefa de implementar um escalonador de URLs, na qual a abordagem utilizada para a resolução foi a implementação de uma estrutura de Fila de Listas com métodos para manipulação dos dados e realização das operações. Com a solução adotada, pode-se verificar o poder dessas estruturas de dados, capazes de resolver problemas complexos através de estruturas relativamente simples. Por meio da resolução desse trabalho, foi possível aplicar e revisar conceitos relacionados a estruturas de dados, alocação de memória, regex, análise de complexidade de tempo e espaço, programação defensiva e performance de programas.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo, entender a especificação, mapear um escalonador para uma estrutura formada por classes, definição das operações e responsabilidades relevantes a cada classe, análise experimental do desempenho do programa, organização do projeto, definição das frentes de trabalho a seguir e a criação de uma documentação completa.

7. Bibliografia

Chaimowicz, L. e 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.

What's the Time Complexity of Average Regex algorithms? Stack Overflow, 2011. Disponível em: < https://stackoverflow.com/questions/5892115/whats-the-time-complexity-of-average-regex-algorithms >. Acesso em: 20 de dez. de 2021.

Input/output file stream class. Cplusplus reference. Disponível em: < https://www.cplusplus.com/reference/fstream/fstream/ >. Acesso em: 10 de dez. de 2021.

Regular Expressions. Cplusplus reference. Disponível em: < https://www.cplusplus.com/reference/regex/ >. Acesso em: 10 de dez. de 2021.

Instruções para compilação e Execução

1 – Extraia o arquivo `.zip` na pasta desejada.

2 – Execute o comando "`cd TP`" no terminal

3 – Execute o comando "`make all`" no terminal para compilar os módulos do programa

4 – Execute o programa `main` da pasta `bin` passando um arquivo de texto com os comandos pela linha de comando. Alguns arquivos de teste estão presentes na pasta tests.