Trabalho Prático 1
Escalonador de URLs

Flávio Marcílio de Oliveira

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
fmo@ufmg.br

1. INTRODUÇÃO

Um dos principais componentes em uma máquina-de-busca é o coletor. Com o auxílio de robôs (também conhecidos como crawlers, spiders), eles varrem a internet e realizam o download do conteúdo apontado por uma URL (o endereço de uma página). Sem perda de generalidade, essas URLs obedecem o seguinte formato:

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

Sabe-se que hoje existem trilhões de URLs e muitas delas apontam para conteúdo inexistente. Visto que a quantidade de recurso é limitada, o coletor necessita de um escalonador para definir a ordem que as páginas apontadas pelas URLs serão coletadas. A ordem depende da estratégia de coleta adotada e as duas mais conhecidas são: *depth-first* (busca em profundidade), que coleta todas as URLs de um host antes de passar para o próximo; e *breadth-first* (busca em largura), que prioriza a variedade e coleta URLs de diferentes hosts simultaneamente, coletando uma URL de cada host.

Este documento apresenta o projeto de um escalonador de URLs utilizando tanto a estratégia *depth-first* quanto a *breadth-first* para escalonar as URLs. Assim, para resolver o problema citado, foi utilizado uma estrutura de dados do tipo FILA para armazenar os hosts na medida que vão sendo conhecidos e para cada host, foi utilizado uma outra estrutura de dados do tipo LISTA para armazenar as URLs. A Fila é ordenada visando priorizar os hosts conhecidos primeiro e as URLs são ordenadas nas Listas seguindo o critério de profundidade e, para aquelas com mesma profundidade, as URLs conhecidas primeiro são inseridas antes das outras.

Este documento está organizado da seguinte forma: Na seção 2 são apresentados os aspectos de implementação, detalhando as estruturas de dados utilizadas, depois é apresentada a forma como o projeto está organizado e por último uma explicação do funcionamento de cada função implementada. Na seção 3 é apresentada as análises de complexidade tanto de tempo quanto de espaço. Na seção 4 são apresentadas as estratégias que garantem a robustez do código. Na seção 5 são apresentadas as análises experimentais realizadas quantificando o desempenho quanto ao tempo de execução e quanto a localidade de referência no uso da memória. Por fim, a seção 6 apresenta as conclusões obtidas com o trabalho. No apêndice A são apresentadas as informações para compilação e execução do programa. No apêndice B é

apresentado o arquivo para os testes de localidade de referência caso se queira replicar os testes desenvolvidos neste trabalho.

2. IMPLEMENTAÇÃO

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1. Estrutura de Dados

O projeto do Escalonador foi desenvolvido utilizando três Tipos Abstrato de Dados (TAD): FILA, LISTA e URL. Além destes TADs, foi projetado um objeto Host contendo como atributos uma string para armazenar o host e um ponteiro para Lista de URLs para armazenar as URLs deste host. A implementação dos TADs FILA e LISTA foram baseados nos slides da aula[1] com suas respectivas operações. O TAD URL foi desenvolvido para facilitar o processamento das URLs. A seguir são apresentadas as características dos TADs com suas respectivas operações e, por fim, como o Escalonador foi desenvolvido.

2.1.1. TAD - Fila

Filas são um TAD com a característica de que o primeiro elemento inserido é o primeiro a ser retirado (FIFO - First In First Out). Esse TAD pode ser implementado seguindo duas abordagens: Sequencial - com uso de arranjos e com alocação estática de memória; ou Encadeada - com uso de apontadores e com alocação dinâmica de memória. Apesar de ter uma implementação mais complexa e uma pior complexidade de tempo para algumas operações, a opção pela abordagem Encadeada se deve por permitir um uso mais interessante da memória e uma liberdade quanto à quantidade de elementos armazenados, o que é fundamental para a aplicação desenvolvida.

Sendo assim, o TAD foi projetado com uma “célula-cabeça” antes do primeiro elemento com o objetivo de reduzir a complexidade de algumas operações e, cada célula com apontadores para as células seguintes, além de um apontador para o objeto que será armazenado na Fila. O TAD contém, ainda, um atributo ***length_*** para armazenar a quantidade de elementos na Fila e dois apontadores: ***head_*** apontando para a “célula-cabeça” e ***tail_*** apontando para o último elemento da Fila. O diagrama esquemático do TAD é apresentado na Figura 1.

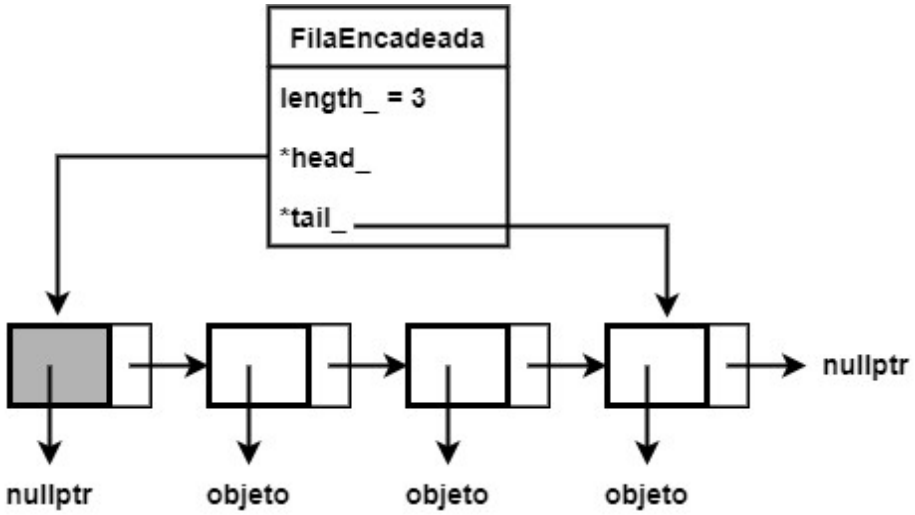


Figura 1 - Diagrama da Estrutura de Dados Fila Encadeada

O TAD possui o método ***Construtor***, responsável por inicializar uma nova Fila criando a "célula-cabeça" e inicializando o atributo ***length_*** com zero, e o ***Destrutor*** responsável por fazer a liberação de toda a memória utilizada na Fila. Além dos métodos construtor e destrutor, foram projetados os métodos: ***size*** que retorna o tamanho da Fila; ***empty*** que verifica se a Fila está vazia ou não; ***push_back*** que insere um novo elemento no final da Fila; ***pop_front*** que remove o primeiro elemento da Fila; ***print*** que imprime todos os elementos da Fila; ***clear*** que exclui todos os elementos da Fila.

Para a implementação do TAD foi utilizado o Padrão de Projeto Template permitindo, assim, que a Fila possa ser instanciada para vários tipos de objetos.

2.1.2. TAD - Lista

Listas são um TAD que permite que os elementos sejam acessados, inseridos e removidos em qualquer posição. A implementação desse TAD pode seguir duas abordagens: Sequencial com o uso de arranjos e alocação estática de memória ou Encadeada com uso de apontadores e alocação dinâmica de memória. Seguindo o critério de utilização mais interessante da memória, optou-se pela abordagem Encadeada para o projeto e implementação da Lista neste trabalho.

O TAD Lista tem seu projeto igual ao projeto do TAD Fila (Figura 1), com os mesmos atributos ***length_***, ***head_*** e ***tail_***. Entretanto, contém métodos adicionais que permitem uma manipulação mais ampla dos elementos. Assim, esse TAD possui os métodos ***Construtor*** responsável por criar a "célula-cabeça" e inicializar o atributo ***length_*** com zero, ***Destrutor*** para desalocar a memória utilizada, ***size*** que retorna a quantidade de elementos da Lista, ***empty*** que verifica se a Lista está vazia ou não, ***get_item*** que retorna o elemento de uma posição específica, ***set_item*** que substitui o elemento de uma posição. Para inserção de novos elementos, o TAD Lista possui três métodos: ***push_front*** que insere um novo elemento na primeira posição da Lista, ***push_back*** que insere um novo elemento no final da Lista e ***insert_at*** que insere um novo elemento em uma posição específica. Para remoção de

elementos também há três métodos: ***pop_front*** que remove o elemento da primeira posição, ***pop_back*** que remove o elemento do final da Lista e ***remove_at*** que remove o elemento de uma posição específica. Ainda há o método ***find*** que pesquisa se um elemento está na Lista e retorna a posição desse elemento caso esteja na lista. Há ainda os métodos ***print*** que imprime todos os elementos da Lista e ***clear*** que remove todos os elementos da Lista.

Para a implementação desse TAD também foi utilizado o Padrão de Projeto Template e, com isso, tornando seu uso mais abrangente.

2.1.3. TAD - Url

Esse TAD foi implementado como uma classe com os seguintes atributos: ***depth_*** para guardar a profundidade da URL, ***protocol_*** para armazenar o protocolo, ***host_*** para armazenar o host, ***path_*** para armazenar o path e ***query_*** para armazenar a query da URL. Como métodos, foram implementados dois ***Construtores***: um padrão para instanciar uma URL com todos os atributos vazios e outro que recebe uma string e processa essa string separando todos os componentes da URL. Foram implementados, ainda, os métodos ***print*** para imprimir a URL, ***GetProtocol*** para retornar o protocolo da URL, ***GetHost*** para retornar o host da URL, ***GetPath*** para retornar o path e ***GetDepth*** para retornar a profundidade da URL. Também foi implementado um método ***isValid*** para verificar se a URL é válida seguindo os requisitos do problema.

Para permitir que essa classe seja utilizada como um tipo de dado para instanciar uma Lista os ***operadores*** << (enviar uma URL para o fluxo de saída de dados) e == (comparar duas URLs) foram sobrecarregados.

2.1.4. Escalonador

O Escalonador possui um ponteiro para uma Fila Encadeada de Host - struct com uma string para armazenar o host da URL e um ponteiro para Lista Encadeada para armazenar as URLs do host - e um atributo ***output_*** para armazenar o nome do arquivo de saída dos resultados. Um diagrama do projeto do Escalonador é apresentado na Figura 2.

Como métodos foram implementados o ***Construtor*** que inicializa o escalonador e o ***Destrutor*** para desalocar a memória utilizada no Escalonador. Além desses, também foram implementados os métodos: ***setOutput*** para configurar o nome do arquivo de resultados, ***AddUrl*** para adicionar novas URLs ao escalonador, ***EscalonaTudo*** para escalonar todas as URLs presentes no escalonador usando a estratégia *depth-first* e ***EscalonaTudoBreadth*** para a estratégia *breadth-first*, ***Escalona*** que escalona uma quantidade definida de URLs usando a estratégia *depth-first* e ***EscalonaBreadth*** para a estratégia *breadth-first*, ***EscalonaHost*** que escalona uma quantidade limitada de URLs de um determinado host, ***VerHost*** que exibe todas as URLs de um determinado host, ***ListaHosts*** que exibe todos os hosts seguindo a ordem em que foram conhecidos, ***LimpaHost*** que exclui todas as URLs de um determinado host e ***LimpaTudo*** que exclui todos os hosts e URLs presentes no Escalonador. Ainda foi implementado um método auxiliar ***PesquisaHost*** que retorna a Lista de URLs de um determinado host para facilitar algumas operações do Escalonador.

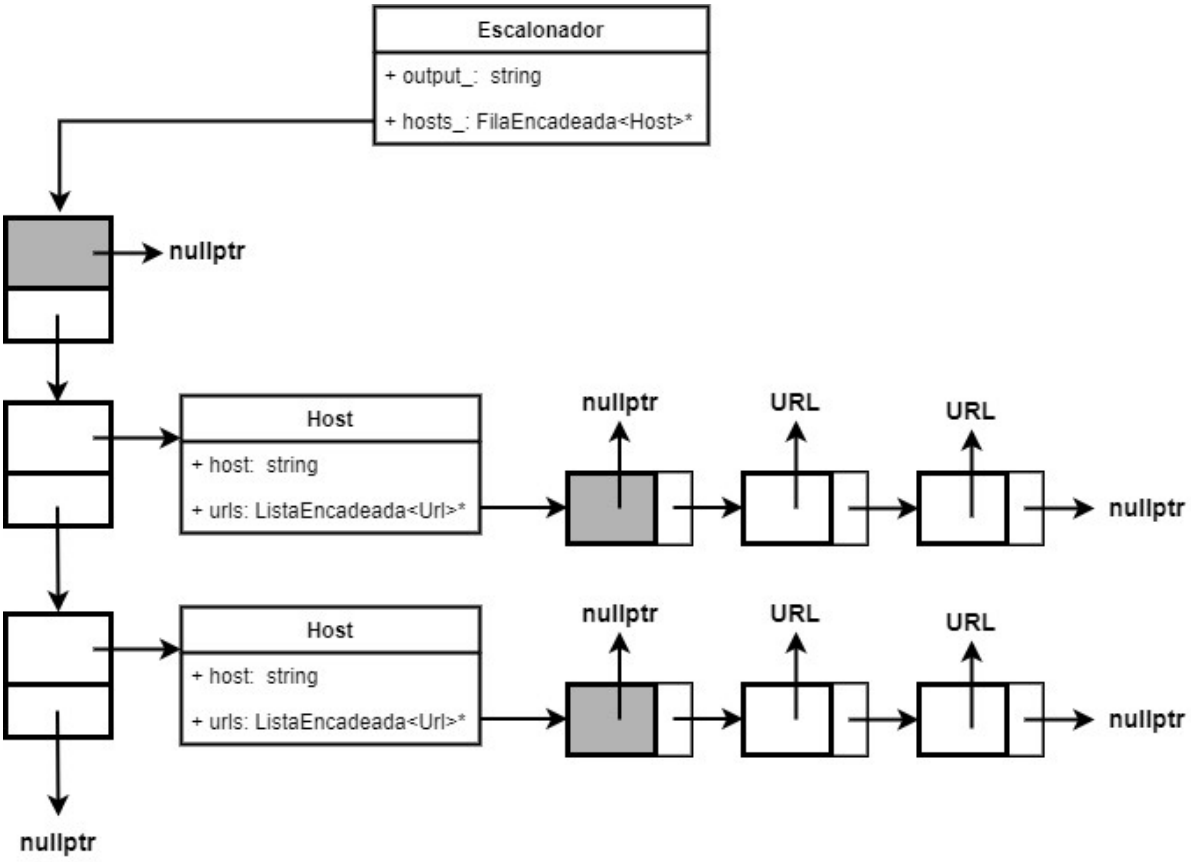


Figura 2 - Diagrama do Escalonador

2.2. Organização do Código

O projeto foi separado em diretórios para melhor organização. Na raiz do projeto está apenas o Makefile responsável por fazer a compilação de todo o projeto. Na pasta **bin** estará o executável do programa gerado após a compilação. Na pasta **include** estão os arquivos de cabeçalho . Na pasta **obj** ficarão os arquivos gerados para linkar na compilação do projeto e gerar o executável. Por fim, na pasta **src** estão os arquivos de implementação .

2.3. Funcionamento do Programa

Para uma melhor compreensão do projeto, apresento uma descrição do funcionamento de cada função do Escalonador e da função principal responsável por ler o arquivo de entrada e chamar as funções específicas do Escalonador.

2.3.1. Operações do Escalonador - escalonador.cpp

Construtor - responsável por alocar uma FilaEncadeada com um tipo de objeto ***Host*** e configura o nome de arquivo de resultados default como “saida-out.txt” (para evitar qualquer tipo de erro no acesso ao arquivo).

Destrutor - responsável por fazer a desalocação de memória. Caso haja elementos na Fila, é chamado o método ***LimpaTudo*** e, por fim, é liberada a memória que contém a Fila.

setOutput - Recebe uma string com o nome do arquivo de saída e configura o nome do arquivo onde os resultados serão armazenados.

AddUrl - Recebe uma URL e pelo host pesquisa na Fila se esse host está presente, se não, aloca uma nova lista de URLs e insere a URL recebida, configura um objeto do tipo Host com o host e com a Lista criada e, por fim, insere na Fila. Caso um elemento com o host pesquisado é encontrado na Fila é feita uma verificação se há elementos na Lista de URLs do host e se não, é feita a inserção. Se na Lista há elementos compara-se se a URL já está presente e se não estiver, é inserida na posição definida pela comparação de profundidade das URLs, de forma que a Lista fique ordenada pela profundidade, ou que não seja inserida uma mesma URL.

EscalonaTudo - A Fila é percorrida por toda a sua extensão retirando o primeiro elemento e, da lista de URLs de cada objeto Host, são retiradas as URLs e impressas. Depois que a Lista é percorrida, o objeto Host com a Lista de URLs vazia é inserido novamente na Fila. Depois que todos os elementos passam por este processo, a Fila está como foi construída originalmente, com todos os seus elementos nas suas respectivas posições, porém, com apenas os hosts conhecidos.

EscalonaTudoBreadth - Os elementos da Fila são retirados um a um e novamente inseridos na Fila para que no final seja possível listar todos os hosts que foram conhecidos. Entre a retirada e a inserção do elemento da Fila, uma URL é retirada deste host. Este processo é repetido até que não reste mais nenhuma URL em nenhum host.

Escalona - Essa operação é responsável por escalonar uma quantidade específica de URLs e para executá-la a estratégia é a mesma do método ***EscalonaTudo***. Na medida em que as URLs do primeiro host vão sendo escalonadas a quantidade vai sendo decrementada. Se o tamanho da Lista de URLs do primeiro host é insuficiente, o elemento Host é colocado novamente na Fila e uma nova Lista é retirada da Fila. Quando a quantidade de URLs é escalonada, os elementos da Fila continuam sendo retirados até que sua configuração esteja como originalmente criada.

EscalonaBreadth - Essa operação segue a mesma ideia da função ***EscalonaTudoBreadth***, porém é utilizado um contador para verificar a quantidade de URLs escalonadas. Quando a quantidade de URLs escalonadas atinge o limite passado ou quando não há mais URLs a serem escalonadas o processo termina.

EscalonaHost - Primeiro é feita a pesquisa do host pela função auxiliar ***PesquisaHost*** que retorna a Lista de URLs deste host. Com essa Lista, as URLs são escalonadas até a quantidade passada como parâmetro ou até o tamanho da Lista se este for menor que a quantidade passada.

VerHost - Com a função auxiliar ***PesquisaHost*** é verificado se há o host na Fila e caso exista, a Lista retornada é percorrida utilizando a função ***get_item*** da Lista e a URL é impressa.

ListaHosts - Todos os elementos da Fila do escalonador são retirados e o host de cada elemento é impresso e, logo em seguida, inserido novamente na Fila. Ao final do processo, a Fila está novamente na sua configuração original.

LimpaHost - Utiliza-se a função auxiliar ***PesquisaHost*** para retornar a Lista de URLs do host e em seguida faz-se a desalocação desta Lista.

LimpaTudo - A Fila do escalonador é percorrida retirando cada elemento e para cada elemento retirado é feita a desalocação da Lista de URLs.

PesquisaHost - Função auxiliar para pesquisar se um determinado host está presente na Fila. Para a execução desta operação, os elementos são retirados da Fila e, para cada elemento retirado é verificado se o host é igual ao pesquisado, se for igual, o endereço da Lista de URLs deste host é copiado para uma variável que será retornada pela função. Caso não seja encontrado o host, o ponteiro para Lista é retornado como nullptr. Depois da comparação, os elementos são novamente inseridos na Fila, de forma a manter a configuração da Fila.

2.3.2. Código principal - main.cpp

Na função principal é instanciado um escalonador e obtido o nome do arquivo de entrada passado na linha de comando. É gerado o nome do arquivo de saída e configurado no escalonador. Com o arquivo de entrada aberto cada linha é lida e passado para uma variável denominada comando. Depois é feita uma comparação com os possíveis comandos que podem ser executados pelo escalonador e, assim, cada função do escalonador é chamada. Apenas o comando "ADD_URLS" tem um passo adicional que é executado depois que a URL é lida do arquivo de entrada e processada pelo TAD URL. Antes de adicionar no escalonador é verificado se a URL é válida e só depois é adicionada caso atenda os critérios de validade.

3. ANÁLISE DE COMPLEXIDADE

3.1. Complexidade de Tempo

A complexidade de tempo foi calculada considerando que as operações de atribuição, de alocação e desalocação de memória e escrever na tela é **O(1)**.

3.1.1. TAD - Fila Encadeada

Construtor: Faz apenas duas atribuições e aloca memória para a "célula-cabeça", portanto, é **O(1)**;

Destrutor: Para uma Fila com *n* elementos é necessário percorrer todas as células fazendo a desalocação da memória (função ***clear***), assim o Destrutor é **O(n)**;

size: Apenas retorna um valor, sendo **O(1)**;

empty: Faz apenas uma comparação e retorna o resultado, sendo **O(1)**;

push_back: Esta função é **O(1)** pois, tem-se apenas duas alocações de memória, quatro atribuições e uma operação de incremento independente do tamanho da Fila;

pop_front: Esta função também é **O(1)** pois todas as operações realizadas tem complexidade assintótica constante;

print: Função com complexidade **O(n)** para uma Fila com *n* elementos, tendo que percorrer toda a Fila para imprimir os elementos;

clear: Complexidade **O(n)** com desalocação de memória de todos os *n* elementos presentes na Fila.

3.1.2. TAD - Lista Encadeada

Construtor: Apenas atribuições e um alocação de memória, portanto, **O(1)**;

Destrutor: Como é necessário percorrer todos as *n* células para fazer a desalocação, tem uma complexidade **O(n)**;

size e **empty**: Apenas operações **O(1)** e, portanto, são funções **O(1)**;

get_item: Pesquisa e retorna um item pela posição, tendo o melhor caso para o elemento na primeira posição realizando apenas uma comparação, portanto, **Ω(1)**. No pior caso, o elemento está na última posição e, neste caso, a complexidade de tempo da função é **O(n)**;

set_item: Esta função segue o comportamento da função **get_item()** acima. No melhor caso é **Ω(1)** e no pior caso **O(n)**;

push_front: função **O(1)** com apenas duas alocações de memória e atribuições;

push_back: função também **O(1)**;

insert_at: Melhor caso ocorre quando a posição escolhida é a primeira, sendo neste caso, **Ω(1)**. No pior caso, a posição escolhida é a última tendo que percorrer toda a Lista, portanto, **O(n)**;

pop_front: Faz apenas operações **O(1)**, como manipulação de ponteiros, portanto, é uma função **O(1)**;

pop_back: É necessário posicionar um ponteiro auxiliar no elemento anterior ao último e, para isso, a Lista é percorrida até a posição *n-1*, portanto, é uma função **O(n)**;

remove_at: No melhor caso a posição escolhida é a primeira, sendo **Ω(1)** e no pior caso a posição é a última sendo **O(n)**;

find: No melhor caso o elemento pesquisado é o primeiro, assim **Ω(1)**. Caso o elemento seja o último ou não esteja na Lista têm-se o pior caso com complexidade **O(n)**;

print e **clear**: Como a Lista é percorrida na sua integralidade em ambas as funções, então são **O(n)**;

3.1.3. TAD - Url

Construtor padrão: Apenas atribuições, portanto **O(1)**;

Construtor com parâmetro: Este construtor é utilizado para processar as URLs passadas como parâmetro do tipo string. O processamento das URLs é realizado com a utilização de funções da biblioteca string do C++, utilizando funções **O(n)**, conforme especificado na documentação. Portanto, este construtor tem complexidade de tempo **O(n)**;

isValid: Faz-se a comparação dos atributos com as especificações do projeto. A comparação é realizada utilizando a função ***compare*** da biblioteca string que é **O(n)**. Portanto, a função isValid é **O(n)**;

print: função **O(1)** pois faz apenas três verificações e depois imprime a URL;

operador<<: complexidade **O(1)**;

operador==: complexidade **O(n)** pois utiliza a função ***compare*** da biblioteca string;

GetProtocol, **GetHost**, **GetPath** e **GetDepth**: são todas **O(1)** pois apenas retornam um valor.

3.1.4. Escalonador

A análise de complexidade do Escalonador foi feita considerando a utilização do escalonador para trabalhar com uma quantidade Q de URLs. Assim, este problema pode aparecer em três cenários: 1) todas as URLs do mesmo host, assim tem-se um Escalonador com 1 host e n URLs, onde $Q = n$; 2) todas as URLs de hosts diferentes, assim tem-se um escalonador com m hosts e 1 URL por host, onde $Q = m$; e 3) m hosts e cada um com n URLs, ou seja, $Q = mn$.

Construtor: Aloca memória para uma nova Fila e faz uma atribuição para todos os cenários listados, portanto, complexidade **O(1)**;

Destrutor: para avaliar a complexidade do destrutor, os três cenários devem ser avaliados pois faz uma chamada à função ***LimpaTudo***, portanto, tem-se:

- Cenário 1 - $Q = n$ (n URLs do mesmo host): a chamada da função ***LimpaTudo*** gera um custo **O(n)** e depois mais uma desalocação da "célula-cabeça" da Fila com custo **O(1)**, portanto, complexidade **O(n)**;
- Cenário 2 - $Q = m$ (m hosts com uma única URL): pela chamada da função ***LimpaTudo*** **O(m)** e **O(1)** pela desalocação da "célula-cabeça" da Fila, assim, complexidade de **O(n)**;
- Cenário 3 - $Q = mn$ (m hosts com n URLs em cada): a chamada da função LimpaTudo gera um custo de $\mathbf{O(n^2)}$ e a desalocação da "célula-cabeça" gera um custo **O(1)**, portanto, complexidade $\mathbf{O(n^2)}$;

Portanto, função com melhor caso com complexidade linear **Ω(n)** para os cenários 1 e 2, e pior caso no cenário 3, com complexidade quadrática $\mathbf{O(n^2)}$.

setOutput: Faz apenas uma atribuição independente do cenário, portanto, **O(1)**;

AddUrl: esta função é implementada de forma a evitar duplicatas e que as URLs sejam inseridas em ordem crescente de profundidade. Assim, além dos cenários apresentados, deve-se avaliar as probabilidades de encontrar uma URL já inserida e também a probabilidade de encontrar uma URL com profundidade maior em uma determinada posição, para então fazer a inserção desta nova URL nesta posição específica. Por ter estes critérios de inserção, a análise da complexidade desta função é mais difícil e exige uma análise probabilística. Portanto, para esta análise será considerado apenas os casos extremos.

- Cenário 1 - $Q = n$ (n URLs do mesmo host): pela chamada inicial da função ***PesquisaHost*** tem-se um custo de **O(1)** em todos os casos. Para inserir uma URL de host diferente tem-se um custo **O(1)**. Para inserir uma URL do mesmo host, tem-se duas situações extremas: inserir na primeira posição ou na última. Se for para inserir na primeira posição, tem-se uma chamada da função ***get_item*** com custo **O(1)** e uma chamada da função ***insert_at*** com custo **O(1)**. Se for inserir na última posição, tem-se n chamadas da função ***get_item*** com um custo crescendo para cada chamada, ou seja, $1 + 2 + \cdots + n = n(n+1)/2$ de forma que o custo total seja $\mathbf{O(n^2)}$ e uma chamada a função ***insert_at*** com custo **O(n)**. Portanto, neste cenário, tem-se o melhor caso para inserir uma URL com host diferente ou inserir no mesmo host na primeira posição com complexidade **O(1) + O(1) = O(1)**, portanto, **Ω(1)** e pior caso para inserir no mesmo host e na última posição com complexidade $\mathbf{O(1) + O(n^2) + O(n) = O(n^2)}$;
- Cenário 2 - $Q = m$ (m hosts com uma única URL cada): inicialmente a função ***PesquisaHost*** é chamada gerando um custo **O(m)**. Para inserir uma URL de host diferente tem-se um custo **O(1)** das funções ***push_back*** da Lista e da Fila. Para inserir uma URL de um host conhecido, tem-se um custo **O(1)** pois a Lista só tem um elemento. Neste cenário, a complexidade é **O(n)** determinada pelo custo da função ***PesquisaHost***;
- Cenário 3 - $Q = mn$ (m hosts com n URLs em cada host): pela chamada da função PesquisaHost tem-se um custo de **O(m)**. Novamente, para inserir uma URL de host diferente tem-se um custo **O(1)**. Para inserir uma URL de host conhecido tem-se a situação já discutida no cenário 1, para inserção no início da Lista o custo é de **O(1)** e para inserção no final da Lista o custo é $\mathbf{O(n^2)}$. Portanto, neste cenário, tem-se o melhor caso para inserir uma URL de host diferente ou inserir no mesmo host na primeira posição com complexidade **O(m) + O(1) = O(m)**, portanto, **Ω(n)** e pior caso para inserir no mesmo host e na última posição com complexidade $\mathbf{O(m) + O(n^2) = O(n^2)}$;

Assim, a função AddUrl apresenta um melhor caso (cenário 1 com inserção da URL com host diferente ou host igual e na primeira posição) com complexidade **Ω(1)** e pior caso (cenário 1 com inserção na última posição e cenário 3 com inserção na última posição de algum host) com complexidade $\mathbf{O(n^2)}$; No cenário 2 e cenário 3 com inserção na primeira posição de um host tem-se uma complexidade **O(n)**, porém não pode ser tratada como uma complexidade

média por não ser evidente que estas situações correspondem a um tipo de entrada "média" para o problema tratado;

EscalonaTudo:

- Cenário 1 - $Q = n$ (n URLs do mesmo host): faz-se uma chamada da função ***pop_front*** da Fila com custo **O(1)**, n chamadas à função ***pop_front*** da Lista com custo **O(1)** por chamada e mais uma chamada à função ***push_back*** da Fila com custo **O(1)**. Assim, tem-se uma complexidade **O(1) + nO(1) + O(1) = O(n)**;
- Cenário 2 - $Q = m$ (m hosts com uma única URL cada): faz-se m chamadas à função ***pop_front*** da Fila com custo **O(1)** para cada uma dessas chamadas, faz-se uma única chamada da função ***pop_front*** da Lista e depois mais uma chamada da função ***push_back*** da Fila com custo **O(1)**. Portanto, neste cenário a complexidade é **m(O(1) + O(1) + O(1)) = O(m)**;
- Cenário 3 - $Q = mn$ (m hosts com n URLs em cada host): faz-se m chamadas às funções ***pop_front*** e ***push_back*** da Fila com custo **O(1)** para cada uma e n chamadas à função ***pop_front*** da Lista com custo O(1) para cada chamada. Assim, tem-se, a complexidade desse cenário dada por **m(O(1) + nO(1) + O(1)) = mO(n) = O(mn) = $O(n^2)$**;

Portanto, esta função tem o melhor caso nos cenários 1 e 2 com complexidade **Ω(n)** e pior caso no cenário 3 com complexidade **$O(n^2)$**;

Escalona: considerando que se queira escalonar uma quantidade suficientemente grande de URLs que no limite são todas as URLs presentes no escalonador:

- Cenário 1 - $Q = n$ (n URLs do mesmo host): tem-se, na execução da função, os seguintes custos: **O(1)** pela chamada da função ***pop_front*** da Fila, n chamadas da função ***pop_front*** da Lista com custo por chamada **O(1)** e uma chamada da função ***push_back*** da Fila (para manter o host no escalonador) com custo **O(1)**. Portanto, neste cenário a função é **O(1) + nO(1) + O(1) = O(n)**;
- Cenário 2 - $Q = m$ (m hosts com uma única URL cada): neste cenário tem-se, uma chamada da função ***pop_front*** da Fila com custo **O(1)** e $m-1$ chamadas das funções ***pop_front*** da Lista, ***pop_front*** e ***push_back*** da Fila todas com custo **O(1)** por chamada. Assim, tem-se uma complexidade de **O(1) + (m-1)(O(1) + O(1) + O(1)) = O(1) + O(m-1) = O(m)**;
- Cenário 3 - $Q = mn$ (m hosts com n URLs em cada host): neste cenário tem-se, m chamadas da função ***pop_front*** da Fila com custo **O(1)** por chamada e em cada uma dessas chamadas, tem-se n chamadas da função ***pop_front*** da Lista com custo **O(1)** por chamada. Assim, tem-se uma complexidade de **m(nO(1) + O(1) + O(1)) = mO(n) = O(mn)**;

Portanto, esta função tem o melhor caso nos cenários 1 e 2 com complexidade **Ω(n)** e pior caso no cenário 3 com complexidade quadrática **$O(n^2)$**.

EscalonaHost: considerando que, no limite, sejam escalonadas todas as URLs do host:

- Cenário 1 - *Q = n* (*n* URLs do mesmo host): chama a função ***PesquisaHost*** com custo **O(1)** e chama a função ***pop_front*** da Lista *n* vezes no máximo com custo **O(1)** por chamada. Portanto, neste caso a função tem complexidade dada por **O(1) + nO(1) = O(n)**;
- Cenário 2 - *Q = m* (*m* hosts com uma única URL cada): chama a função ***PesquisaHost*** com custo **O(m)** e chama a função ***pop_front*** uma única vez com custo **O(1)**. Portanto, neste cenário a complexidade é **O(m) + O(1) = O(m)**;
- Cenário 3 - *Q = mn* (*m* hosts com *n* URLs em cada host): chama a função ***PesquisaHost*** com custo **O(m)** e chama a função ***pop_front*** da Lista *n* vezes no máximo com custo **O(1)** por chamada. Assim, a complexidade neste cenário será **O(m) + nO(1) = O(m) + O(n) = O(n)**;

Portanto, a função *EscalonaHost* tem uma complexidade assintótica **O(n)** em todos os cenários;

VerHost:

- Cenário 1 - *Q = n* (*n* URLs do mesmo host): utiliza a função ***Pesquisahost*** com custo **O(1)** e utiliza a função ***get_item*** da Lista *n* vezes. Como a função ***get_item*** é chamada para todos os elementos da Lista em sequência, ou seja, para o primeiro elemento custo de 1 unidade de tempo, para o segundo 2 unidades de tempo e assim até o elemento na posição *n* com custo de *n* unidades de tempo, assim, o custo total desta parte será dada por

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

  De modo que, neste cenário a complexidade da função **VerHost** é $\mathbf{O(n^2)}$;

- Cenário 2 - *Q = m* (*m* hosts com uma única URL cada): utiliza a função ***PesquisaHost*** com custo **O(m)** e chama a função get_item da Lista apenas uma vez com custo **O(1)**. Assim, neste cenário a função é **O(m)**;
- Cenário 3 - *Q = mn* (*m* hosts com *n* URLs em cada host): utiliza a função ***PesquisaHost*** com custo **O(m)** e utiliza a função ***get_item*** da Lista *n* vezes como no cenário 1. Assim, neste caso a complexidade será $\mathbf{O(n^2)}$ determinada pela chamada da função ***get_item***;

Portanto, esta função tem o melhor caso para o cenário 2 com complexidade **Ω(n)** e pior caso nos cenários 1 e 3 com complexidade $\mathbf{O(n^2)}$;

ListaHosts:

- Cenário 1 - *Q = n* (*n* URLs do mesmo host): acessa apenas um elemento da Fila com custo **O(1)**;

- Cenário 2 - $Q = m$ (m hosts com uma única URL): acessa m elementos da Fila com custo **O(m)**;
- Cenário 3 - $Q = mn$ (m hosts com n URLs em cada): acessa m elementos da Fila com custo **O(m)**;

Portanto, no cenário 1 tem-se o melhor caso com complexidade **Ω(1)**. O pior caso ocorre para os cenários 2 e 3 com complexidade linear **O(n)**;

LimpaHost:

- Cenário 1 - $Q = n$ (n URLs do mesmo host): chama a função ***PesquisaHost*** com custo **O(1)** e depois desaloca a Lista com custo **O(n)**. Portanto, neste cenário a função tem complexidade **O(1) + O(n) = O(n)**;
- Cenário 2 - $Q = m$ (m hosts com uma única URL cada): chama a função ***PesquisaHost*** com custo **O(m)** e depois desaloca apenas um elemento da Lista com custo **O(1)**. Portanto, complexidade neste cenário de **O(m) + O(1) = O(m)**;
- Cenário 3 - $Q = mn$ (m hosts com n URLs em cada host): chama a função ***PesquisaHost*** com custo **O(m)** e depois desaloca uma Lista com custo **O(n)**. Assim, tem-se, **O(m) + O(n) = O(max(m,n)) = O(n)**.

Portanto, esta função tem uma complexidade assintótica linear **O(n)** para os três cenários;

LimpaTudo:

- Cenário 1 - $Q = n$ (n URLs do mesmo host): delete em uma Lista **O(n)**;
- Cenário 2 - $Q = m$ (m hosts com uma única URL): delete em uma Fila **O(m)**;
- Cenário 3 - $Q = mn$ (m hosts com n URLs em cada): para cada elemento da Fila, uma Lista é percorrida, portanto, **O(mn) = O(n^2)**;

Portanto, função com melhor caso com complexidade linear **Ω(n)** para os cenários 1 e 2, e pior caso no cenário 3, com complexidade quadrática **O(n^2)**.

PesquisaHost: Função que percorre apenas elementos da Fila, portanto, tem-se:

- Cenário 1 - $Q = n$ (n URLs do mesmo host): acessa apenas um elemento, portanto, **O(1)**;
- Cenário 2 - $Q = m$ (m hosts com uma única URL cada): acessa os m hosts, para que a Fila permaneça igual, portanto, **O(m)**;
- Cenário 3 - $Q = mn$ (m hosts com n URLs em cada host): percorre apenas os m hosts como no cenário 2, portanto, **O(m)**;

Assim, tem-se o melhor caso no cenário 1 com complexidade **Ω(1)**, e pior caso nos cenários 2 e 3 com complexidade **O(n)**;

3.2. Complexidade de espaço

A análise da complexidade de espaço é apresentada para as Estruturas de Dados implementadas e para o Escalonador considerando um problema de tamanho *n*.

3.2.1. TAD - Fila Encadeada

Para armazenar *n* elementos utilizando o TAD Fila Encadeada são necessárias *n+1* células (a "célula-cabeça" e os *n* elementos) e cada célula armazena dois ponteiros (um ponteiro para o tipo de item a ser armazenado e outro para a próxima célula), portanto, *2(n+1)* unidades de memória para as células e *n* unidades de memória para os elementos. Além disso, o TAD armazena um inteiro para a quantidade de elementos e dois ponteiros para células (***head_*** e ***tail_***). Sendo assim, a quantidade de memória utilizada pode ser dada pela função de complexidade:

$$f(n) = 2(n + 1) + n + 1 + 2 = 3n + 5$$

Portanto, a Fila Encadeada tem uma complexidade de espaço **O(n)**.

3.2.2. TAD - Lista Encadeada

O TAD Lista Encadeada segue o mesmo projeto da Fila Encadeada e, portanto, tem complexidade de espaço **O(n)**.

3.2.3. TAD - URL

O TAD Url armazena cinco strings e um inteiro. Como as strings podem ter qualquer tamanho, dependendo da URL passada, a complexidade desse TAD é **O(n)**.

3.2.4. Escalonador

O projeto do escalonador envolve uma string para armazenar o arquivo de saída e uma Fila Encadeada de *n* células, sendo cada célula da Fila um objeto contendo uma string para armazenar o host e uma Lista Encadeada de *n* URLs. Portanto, a complexidade de espaço para o escalonador é **nO(n) = O(n^2)**.

4. ESTRATÉGIAS DE ROBUSTEZ

Para garantir a tolerância às falhas e facilitar o desenvolvimento de testes de unidade, optou-se por projetar os TADs com a estratégia de lançamento de exceções para tratar as possíveis entradas inválidas. Sendo assim, caso seja passado um parâmetro com valor inválido, a função lança uma exceção específica que pode ser capturada por blocos de código try…catch e, assim, ser tratada por quem chamou a função.

Para desenvolver essa estratégia, foi desenvolvido duas structs de exceções: ***ExcecaoEmpty*** - que lança uma mensagem de exceção se uma função for chamada para alguma operação que não possa ser feita se a Lista ou a Fila estiverem vazias; ***ExcecaoInvalidPosition*** - se uma

posição inválida for passada para ser manipulada nos TADs, neste caso, além da mensagem é informado o intervalo válido e a posição inválida que foi passada.

Utilizando essa estratégia, o TAD Fila Encadeada lança a exceção ***ExcecaoEmpty*** caso as funções ***pop_front***, ***print*** e ***clear*** sejam chamadas com a Fila vazia, evitando assim que porções inválidas de memória sejam acessadas.

No TAD Lista Encadeada, as funções ***get_item***, ***set_item***, ***insert_at*** e ***remove_at*** lançam a exceção ***ExcecaoInvalidPosition*** para evitar posições além dos limites e as funções ***get_item***, ***set_item***, ***insert_at***, ***pop_front***, ***pop_back***, ***remove_at***, ***find***, ***print*** e ***clear*** lançam a exceção ***ExcecaoEmpty*** evitando operações em uma Lista vazia.

5. ANÁLISE EXPERIMENTAL

A análise experimental foi desenvolvida para um conjunto de operações (ADD_URLS, LISTA_HOSTS, ESCALONA_TUDO e LIMPA_TUDO) passados nessa sequência buscando avaliar tanto o desempenho em questão de tempo de execução quanto para a localidade de referência. Os arquivos de testes foram gerados pelo programa disponibilizado pelo professor no Moodle.

As análises apresentadas nesta seção foram realizadas utilizando um computador com as seguintes especificações:

Processador: Intel(R) Core™ i3 CPU M 370 @ 2.40GHz

RAM: 8,00GB

Sistema operacional de 64 bits: Windows 10 Pro versão 21H1

WSL2: Ubuntu-20.04

5.1. Análise de Desempenho

Para a análise de desempenho do programa foram criados cinco casos de testes contendo 114, 214, 314, 414 e 514 URLs geradas pelo programa geracarga disponibilizado. Para os casos de testes foram considerados uma distribuição das URLs em 10 hosts, variância de URLs por host de 5, profundidade média de 5 e variância da profundidade média de 5. Para gerar as URLs foram considerados como número médio de URLs por host como 10, 20, 30, 40 e 50 para cada caso de teste respectivamente.

Os arquivos de testes gerados pelo programa correspondem ao cenário 3 discutido nas análise de complexidade, sendo um número *m* de hosts e *n* de URLs para cada host. Em estudos posteriores, seria interessante fazer uma análise para os outros dois cenários e para as funções separadamente ou para conjuntos diferentes de comandos.

A Figura 3 mostra o gráfico obtido para os cinco casos de testes obtidos.

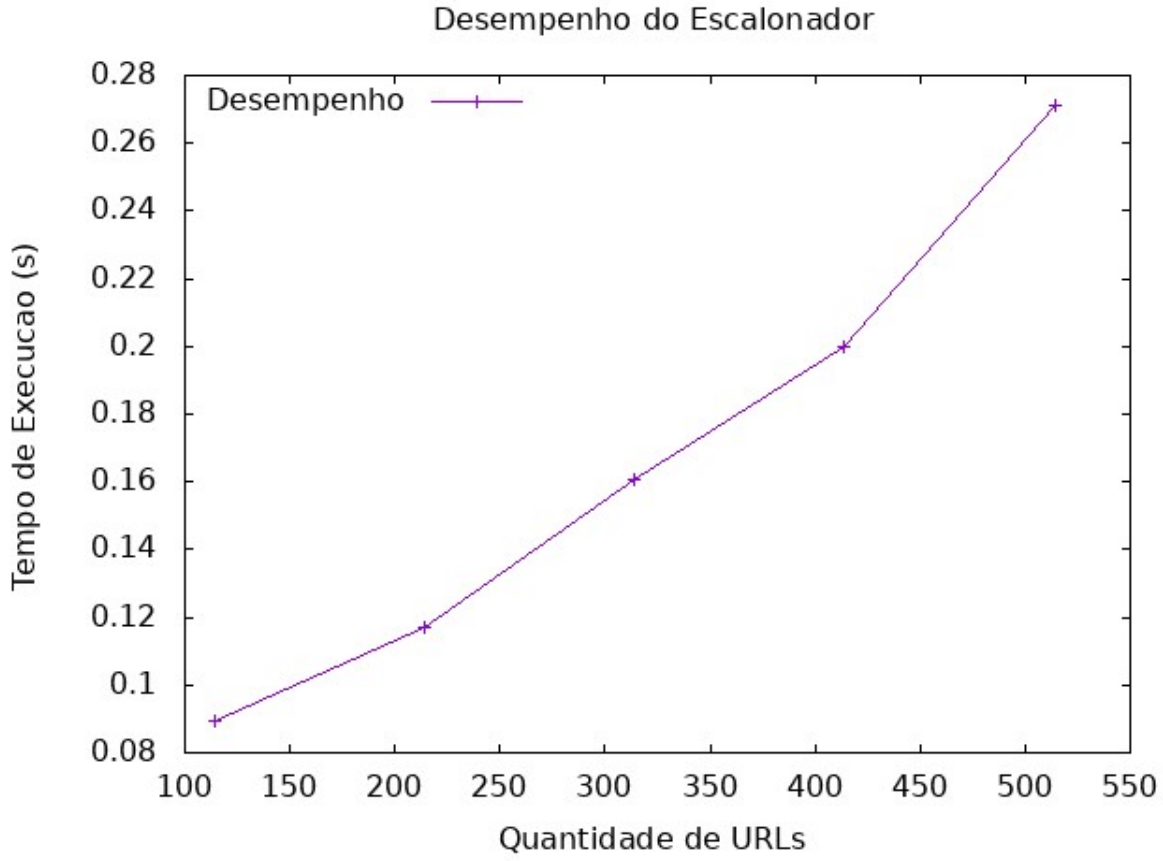


Figura 3 - Desempenho do Escalonador

Nos testes são executadas as funções AddUrl, ListaHosts, EscalonaTudo e LimpaTudo, que como visto na seção 3.1.4, possuem complexidade de tempo, respectivamente, de $O(n^2)$, $O(n)$, $O(n^2)$ e $O(n)$, para a execução na sequência em que são chamadas. Assim, na execução dos testes espera-se que a complexidade de tempo seja quadrática e, pela Figura 3, pode-se observar o comportamento esperado.

5.2. Análise de Localidade de Referência

A Localidade de Referência é a tendência de um processador acessar o mesmo conjunto de locais de memória repetidamente por um período curto de tempo[2], ou seja, em períodos curtos de tempo o acesso à memória tende a ser em endereços próximos. Para fazer a análise da Localidade de Referência foram utilizados o Mapa de Acesso à Memória e a Distância de Pilha.

Como os casos de testes podem ter muitas combinações, envolvendo número de hosts e profundidade das URLs diferentes, foi utilizado um total de 26 URLs divididas em 5 hosts diferentes e com profundidade mínima de 2 e máxima de 6. O arquivo de teste utilizado está disponível no Apêndice.

Para facilitar a análise da Localidade de Referência, é apresentado na Tabela 1, a sequência de acesso à memória de acordo com a distribuição das URLs no arquivo de entrada.

Tabela 1 - Acesso à memória na Inserção das URLs

LISTAS	SEQUÊNCIA DE ACESSOS À MEMÓRIA																								
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
L1	X	X	X						X	X									X							
L2				X	X							X	X			X	X								X	
L3						X	X											X						X		
L4								X			X									X		X				
L5														X	X						X		X			X

Cada Lista corresponde a um host diferente e são criadas no momento em que surge um novo host ainda não presente no Escalonador.

Devido à estratégia de implementação utilizando ponteiros tanto para as células quanto para os elementos das células, foi utilizado uma distinção de IDs para identificar as células da Fila e da Lista bem como para identificar os elementos da Fila e os elementos da Lista. Portanto, para identificar o acesso à memória das diferentes estruturas presentes no programa, foi definido os seguintes IDs:

Tabela 2 - IDs para Mapa de Acesso

IDs	Estruturas
0	Células da Fila
1-5	Células das Listas
100	Elementos da Fila - Host
101-105	Elementos das Lista - URLs

5.2.1. Mapa de Acesso à Memória

A Figura 4 mostra o mapa de acesso para as células da Fila Encadeada.

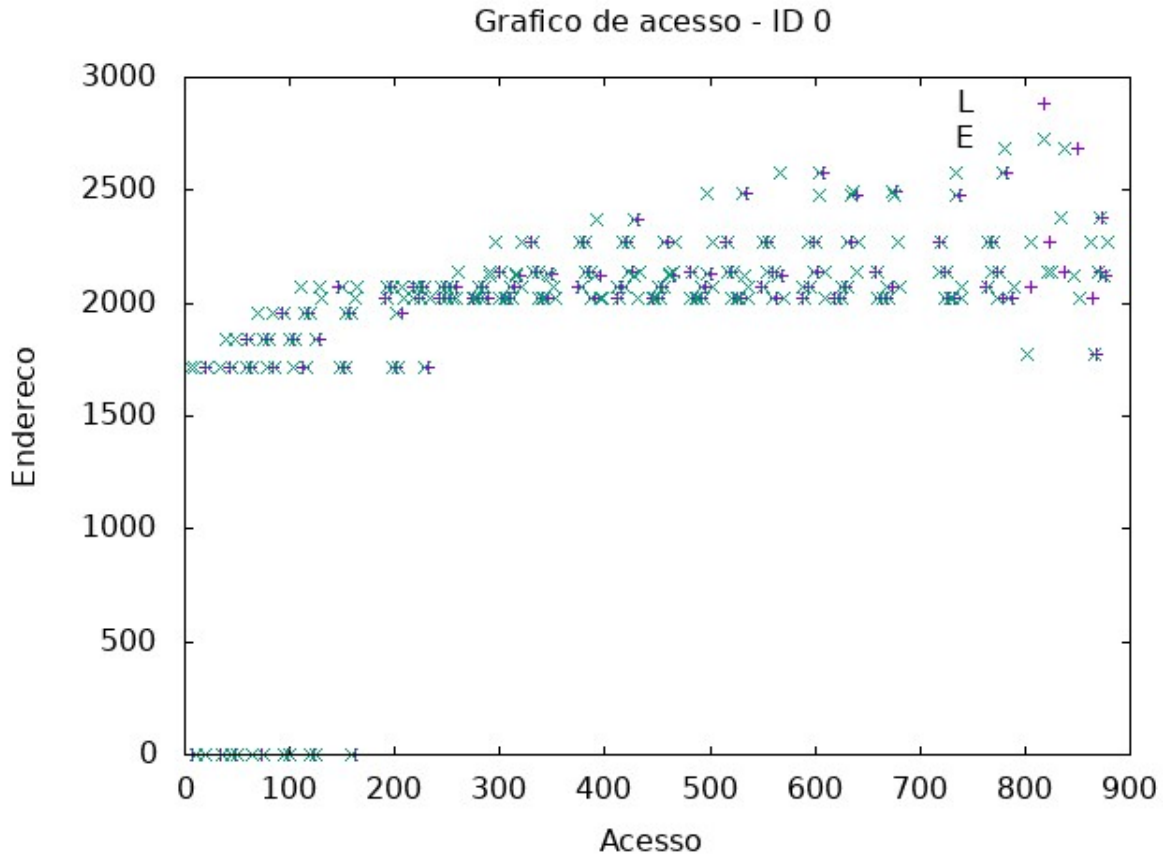


Figura 4 - Mapa de acesso para as células da Fila Encadeada

Observa-se, pela análise do Mapa de Acesso apresentado na Figura 4, o processo de acesso às células da Fila pela função *pesquisaHost*. Como foi feita a opção por não utilizar uma estrutura de dados auxiliar para armazenar temporariamente as células removidas da Fila *hosts_*, pelo contrário, as células são retiradas, feita a comparação para ver se é o host pesquisado e logo em seguida inserida na Fila *hosts_* novamente, o processo de leitura e escrita na Fila gera um número grande de acessos à memória. É possível observar também, que a cada nova célula criada, têm-se uma outra célula (*next_*) com endereço 0 (*nullptr*), o que gera 12 acessos de escrita à células com endereço 0 e somente 7 acessos de leitura, feitos quando a função *pesquisaHost* é chamada.

A Figura 5 mostra o mapa de acesso para os elementos da Fila Encadeada (Host).

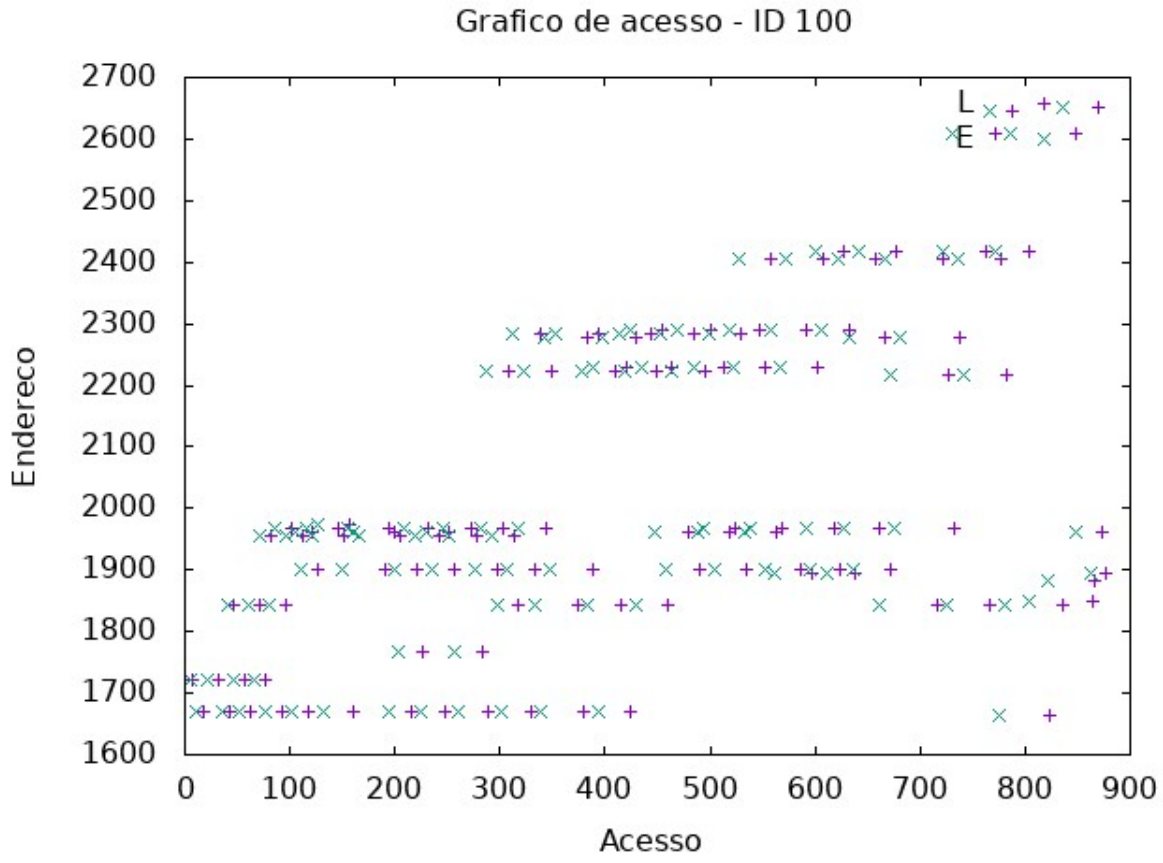


Figura 5 - Mapa de acesso para os elementos (Host) da Fila Encadeada

O mesmo processo de acesso à memória das células da Fila ocorre para os seus elementos, já que na função *pesquisaHost* é feito o uso dos métodos *pop_front* e *push_back* da Fila, o que em ambos, são feitos acessos tanto à célula quanto ao elemento. No método *push_back* temos a escrita na memória da célula e do elemento e no método *pop_front* tem-se o acesso ao elemento para poder retorná-lo e um acesso de leitura para determinar a célula a ser retirada e outro acesso de escrita para mudar o ponteira para a cabeça.

Para realizar os comandos LISTA_HOSTS, ESCALONA_TUDO e LIMPA_TUDO, também são feitas leituras e escritas na Fila, gerando um mapa com tantos acessos como apresentados nas Figuras 4 e 5. Como foi utilizado a estratégia de escalonamento *depth-first*, é possível notar que a realização destas operações gera um processo de leitura e escrita na memória em regiões muito próximas, permitindo fazer uma nítida distinção entre cinco regiões no mapa de acesso apresentado na Figura 5.

Para incluir as URLs no escalonador são criadas Listas Encadeadas diferentes para cada host distinto encontrado. Assim, a Figura 6 mostra o mapa de acesso para as células da Lista 1, ou seja, a Lista criada para as URLs do primeiro host encontrado no arquivo de teste.

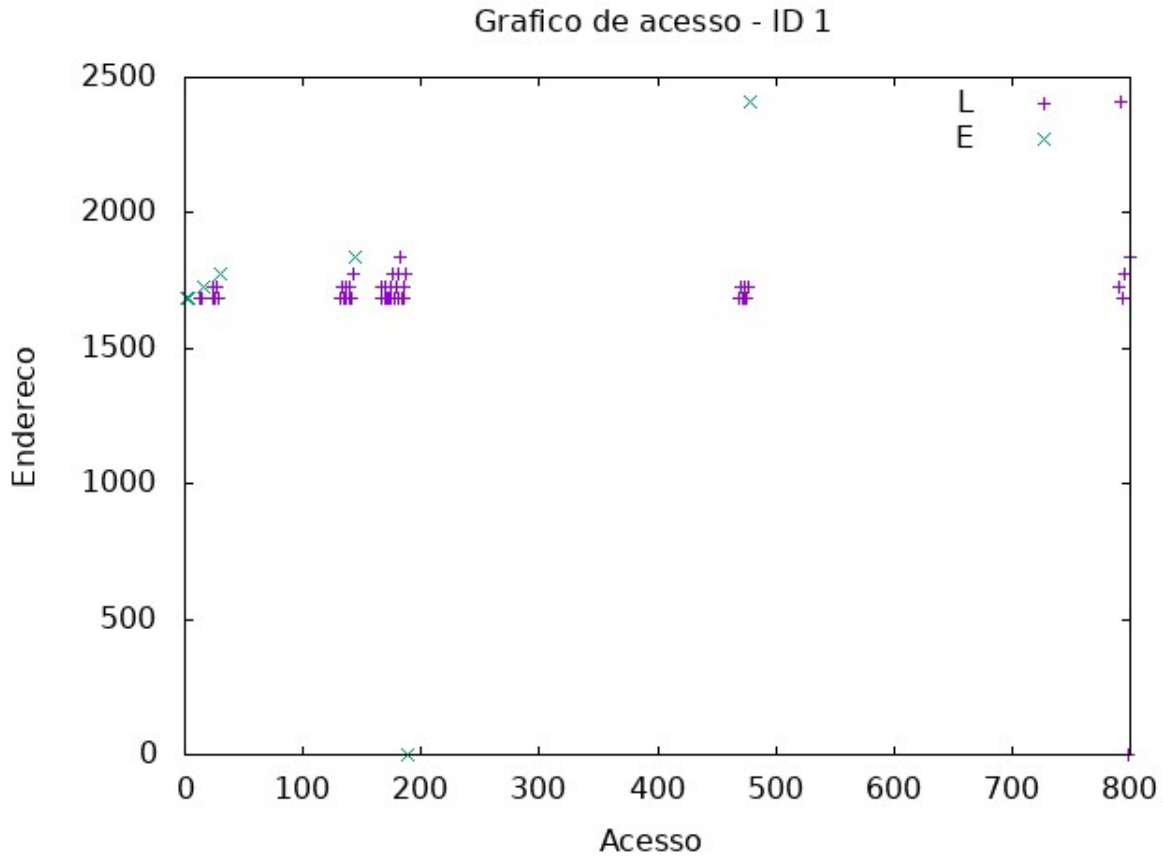


Figura 6 - Mapa de acesso das células da Lista 1

O mapa de acesso à memória para as células da primeira Lista permite entender o funcionamento do programa e como estão distribuídas as URLs no arquivo de entrada. No primeiro acesso quando ainda não há células na Lista é criada a "célula-cabeça" e a célula que vai receber a primeira URL, assim aparecendo no mapa dois acessos de escritas quase sobrepostos. Como o arquivo de entrada fornece três URLs do mesmo host em sequência, vemos no mapa a criação de três células sucessivas. Além disso, como foi implementado uma lógica de verificação em qual posição devem ser inseridas as URLs, a partir da primeira sempre vão ter acessos de leitura e esses acessos acontecem até que uma URL com profundidade maior seja encontrada. Com isso, vemos que para inserir a segunda URL foi necessário ler duas células: primeira leitura correspondente a primeira URL no método *get_item* e a segunda leitura no método *insert_at* para ler a célula-cabeça. Assim, quanto mais URLs na Lista mais acessos serão necessários para verificar em qual posição inserir uma nova URL.

O mapa de acesso para as URLs da primeira Lista é apresentado na Figura 7.

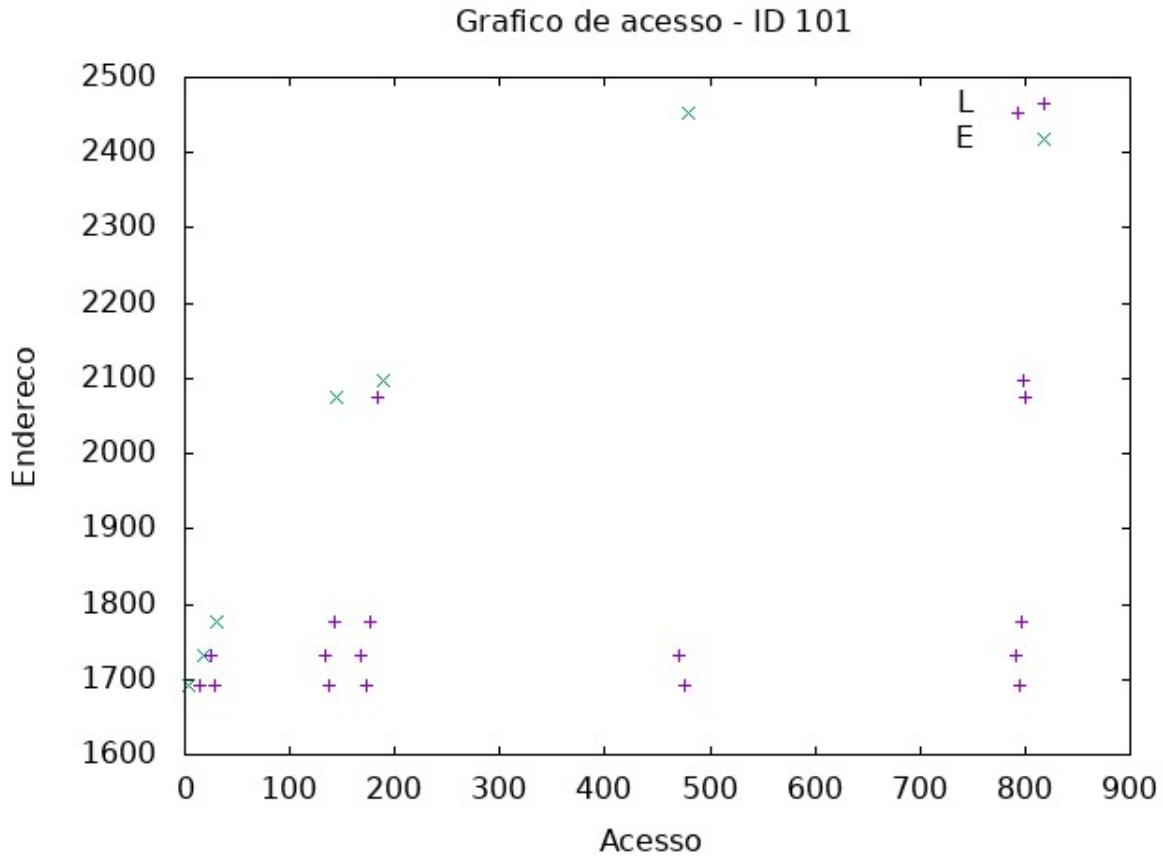


Figura 7 - Mapa de acesso dos elementos (URLs) da Lista 1

Pelo mapa de acesso das URLs da primeira Lista observa-se que a escrita na memória é dividida basicamente em três momentos distintos, confirmando a previsão feita inicialmente e que consta na Tabela 1. Observa-se três acessos de escritas sucessivas em um primeiro momento inserindo as três primeiras URLs lidas do arquivo, em um segundo momento mais dois acessos de escrita e, por fim, a escrita da última URL para este mesmo host. O ponto mais interessante que pode ser observado com o mapa de acesso é o processo de leitura das URLs para permitir a inserção na posição correta. Nota-se que, para inserir a segunda URL (profundidade 2) foi necessário ler a primeira (profundidade 3) e para inserir a terceira URL (profundidade 3) foi necessário ler as duas presentes na Lista. Para inserir a quarta URL (profundidade 5) foi necessário ler as três anteriores pois nenhuma tinha profundidade maior e para inserir a quinta URL (profundidade 4) também foi necessário ler todas as anteriores apesar de a última ser maior mas ainda deve ser lida para efeito de comparação. No caso da inserção da sexta URL (profundidade 2) só foi necessário ler as duas primeiras, com profundidades 2 e 3, descobrindo-se uma URL com profundidade maior e fazendo a inserção, sem percorrer os outros elementos da Lista.

O mapa de acesso para a segunda Lista, correspondente ao segundo host conhecido pelo Escalonador é mostrado na Figura 8, para as células e na Figura 9 para os elementos (URLs).

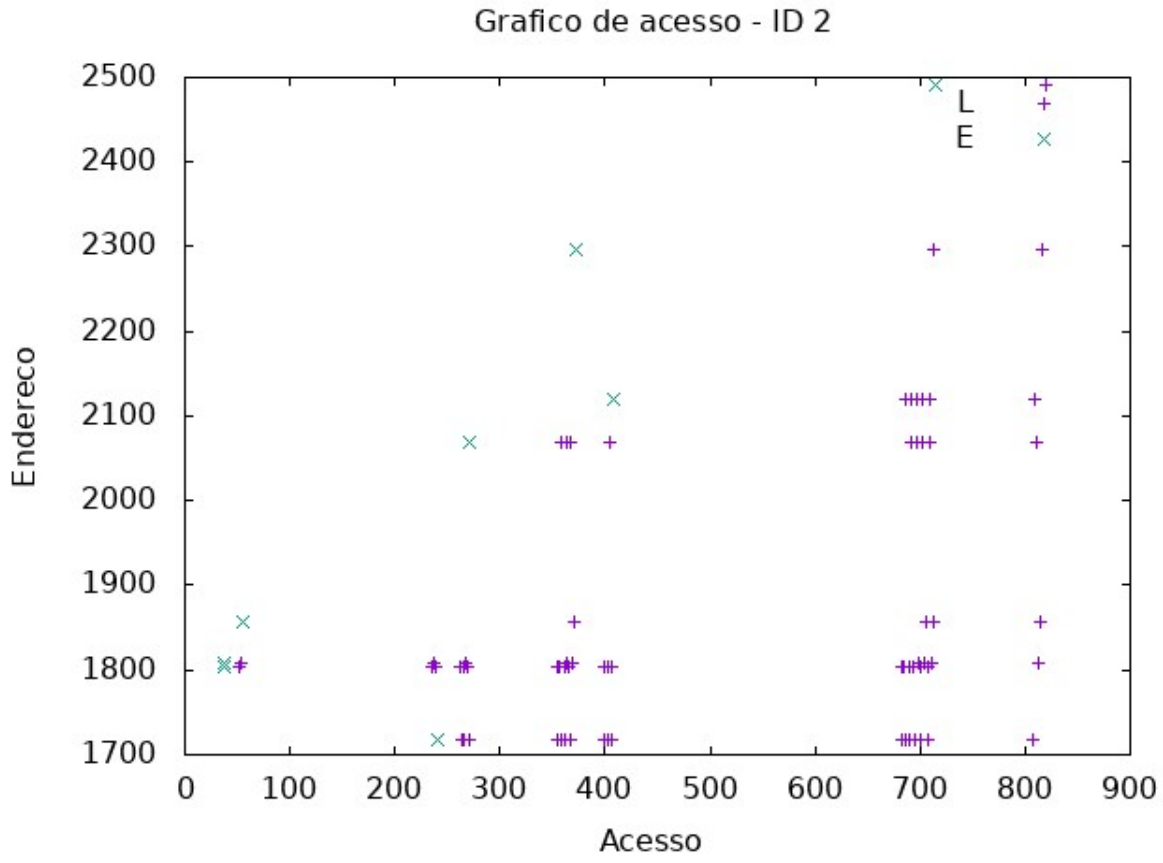


Figura 8 - Mapa de acesso das células da Lista 2

No mapa de acesso para as células da segunda Lista observa-se a escrita de 8 células, sendo 7 para armazenar as URLs e uma para a "célula-cabeça". O comportamento para a primeira escrita se repete, gerando dois acessos quase sobrepostos. Nas demais inserções, nota-se vários acessos de leitura das células para percorrer a Lista e determinar a posição de inserção da nova célula.

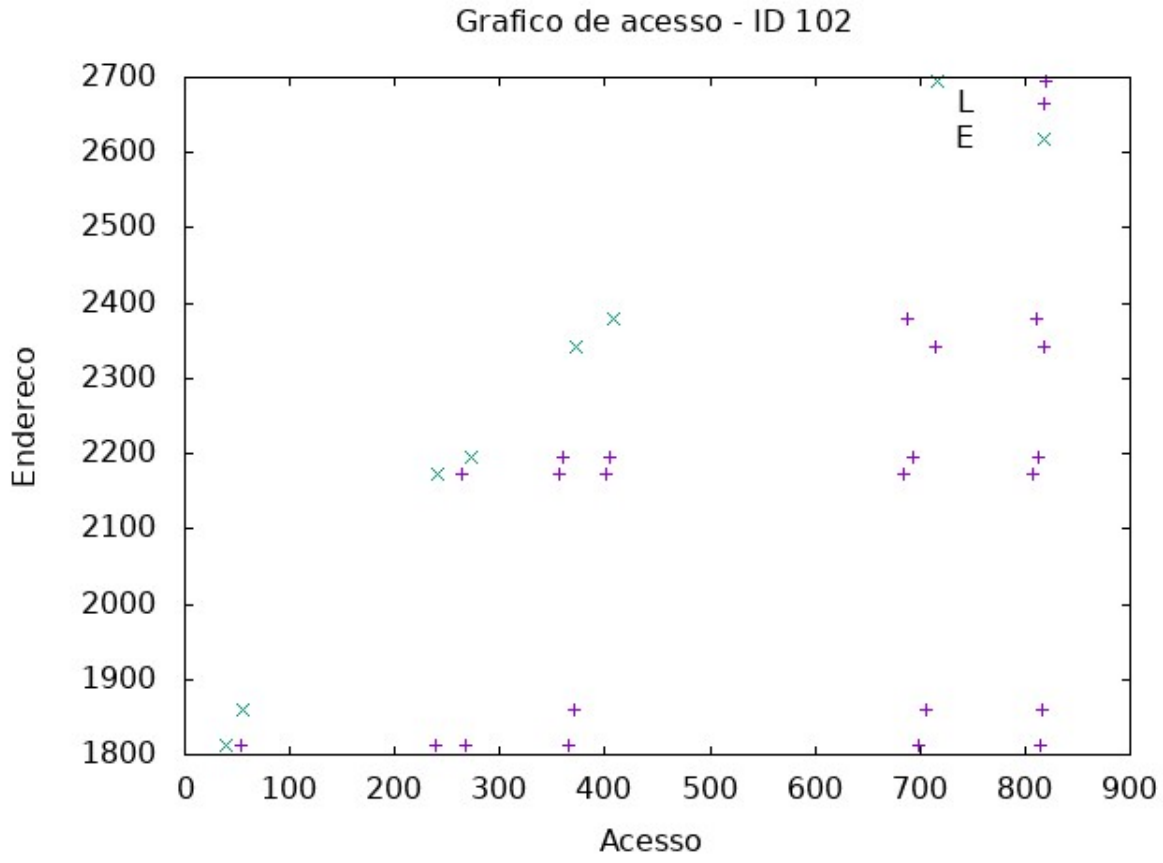


Figura 9 - Mapa de acesso dos elementos (URLs) da Lista 2

Claramente nota-se que são 7 URLs para este host e que elas estão distribuídas no arquivo de entrada em três conjuntos de duas URLs e um URL bem distante das anteriores, conforme mostrado na Tabela 1. Pelo mapa apresentado na Figura 9 observa-se que para inserir a terceira URL (profundidade 2) só foi necessário ler a primeira (profundidade 4), indicando que a profundidade da terceira URL é menor que a que estava na primeira posição. Para inserir a quarta URL (profundidade 3), as duas primeiras da Lista foram lidas indicando que esta nova URL tem uma profundidade maior que a primeira e menor que a segunda. Na inserção da quinta URL (profundidade 5), todas as presentes na Lista foram lidas indicando que esta URL tem um profundidade maior que todas as outras. Para inserir a sexta URL (profundidade 2) só foram lidas as duas primeiras, verificando-se que esta URL tem uma profundidade menor do que a que estava na posição 2. Finalmente, a última URL com profundidade 6 exigiu que fossem lidas todas as que estavam presentes na Lista e assim fosse inserida na última posição.

A terceira Lista apresenta o mapa de acesso para as células mostrado na Figura 10 e o mapa de acesso para as URLs mostrado na Figura 11.

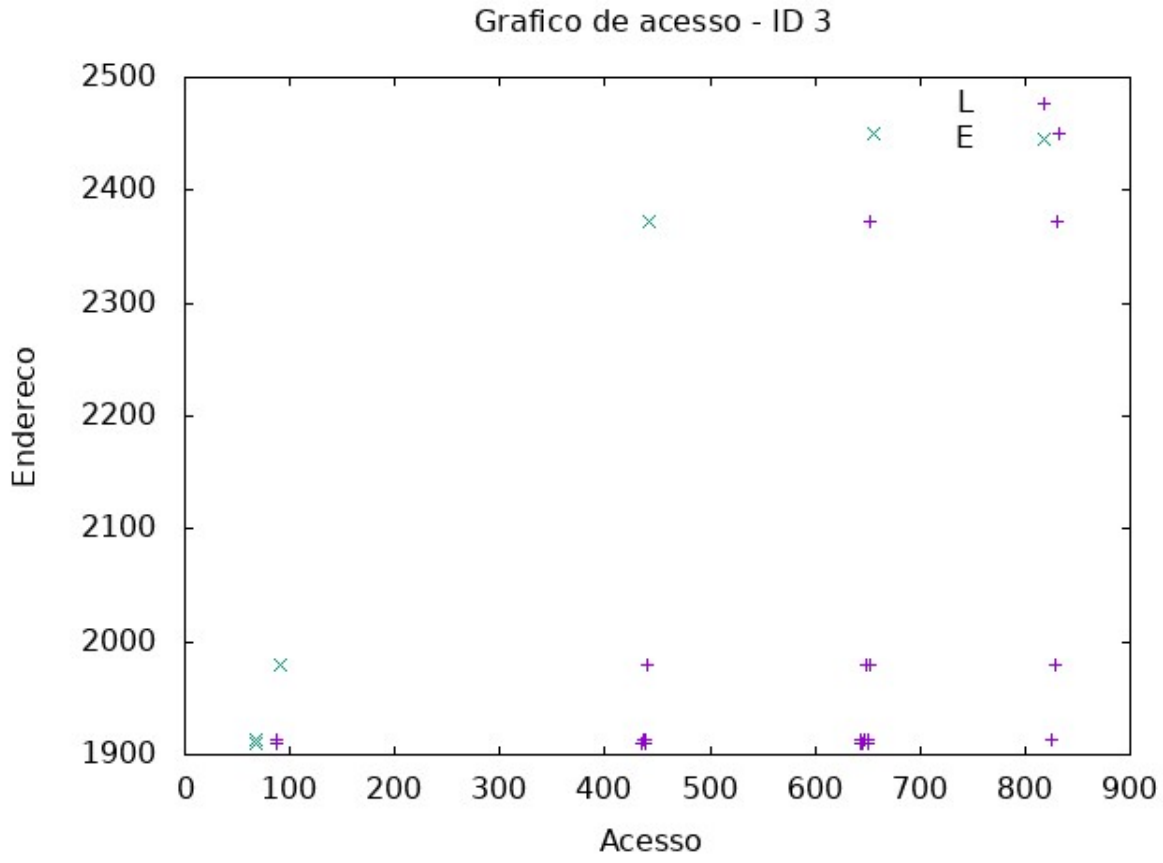


Figura 10 - Mapa de acesso para as células da Lista 3

Pelo mapa de acesso da terceira Lista observa-se que são inseridas quatro URLs, sendo necessárias, para isso, cinco células. O acesso a cada célula é feito no modo leitura antes de ser inserida a nova célula.

Na Figura 11, é apresentado o mapa de acesso para as URLs.

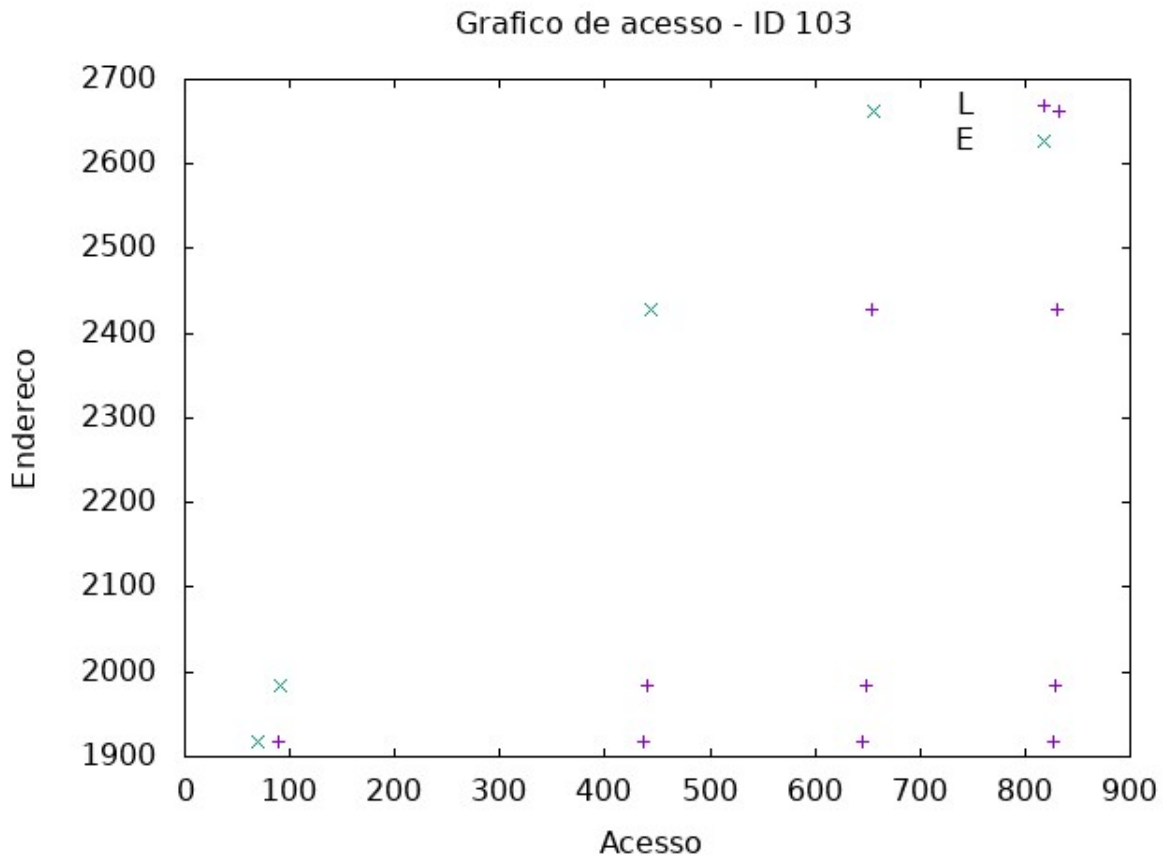


Figura 11 - Mapa de acesso para os elementos (URLs) da Lista 3

Para as URLs inseridas na Lista 3, o mapa de acesso à memória é bem uniforme, mostrando que todas as URLs estão organizadas em ordem crescente de profundidade no arquivo de entrada. Essa ordenação prévia obriga um acesso a todos os elementos presentes na Lista antes de inserir um novo. Por outro lado, se fossem inseridas URLs dispostas no arquivo de entrada em ordem decrescente de profundidade, pode-se concluir que apenas um acesso de leitura seria realizado no processo de inserção de cada uma das URLs.

As Figuras 12 e 13 mostram, respectivamente, o mapa de acesso para as células e elementos da quarta Lista correspondente ao quarto host distinto conhecido pelo Escalonador.

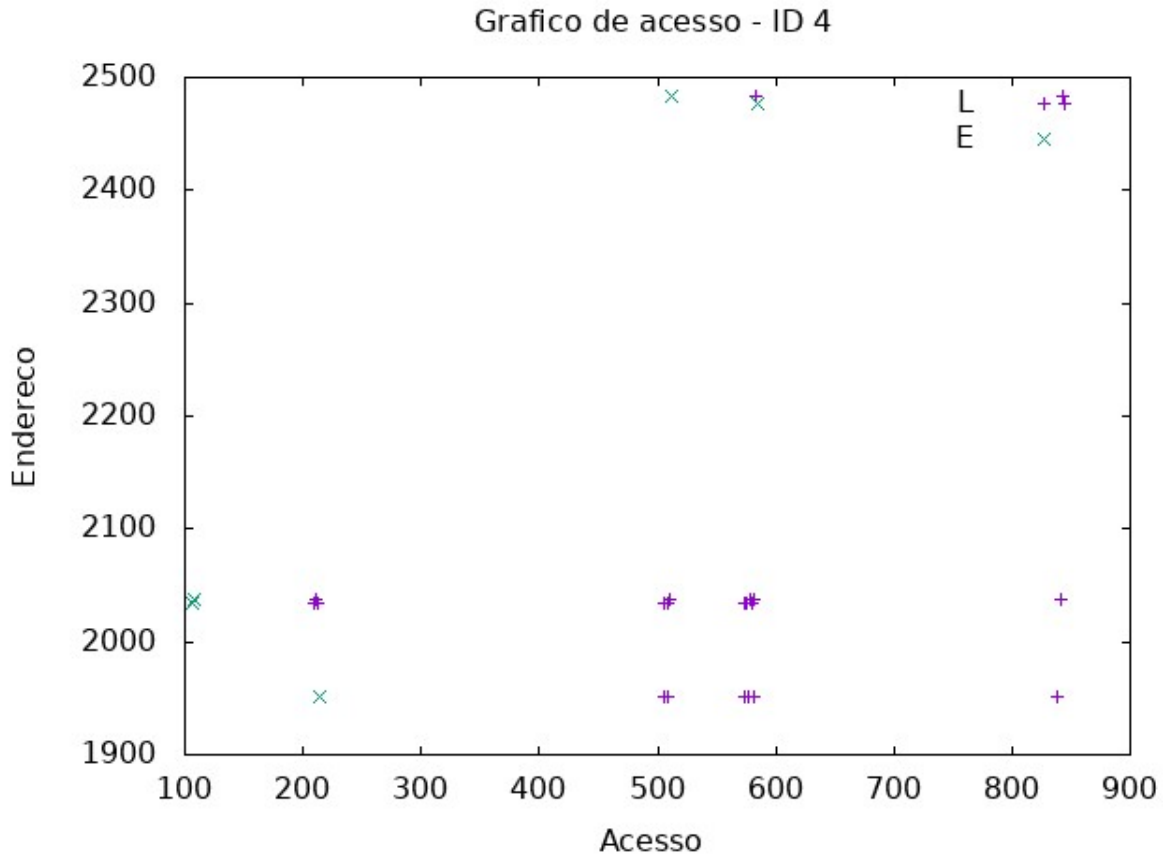


Figura 12 - Mapa de acesso para as células da Lista 4

Como são apresentados cinco endereços de células, conclui-se que são inseridas quatro URLs e, pela dispersão no mapa, conclui-se também que são incluídas em momentos diferentes, ou seja, não estão em sequência no arquivo de entrada.

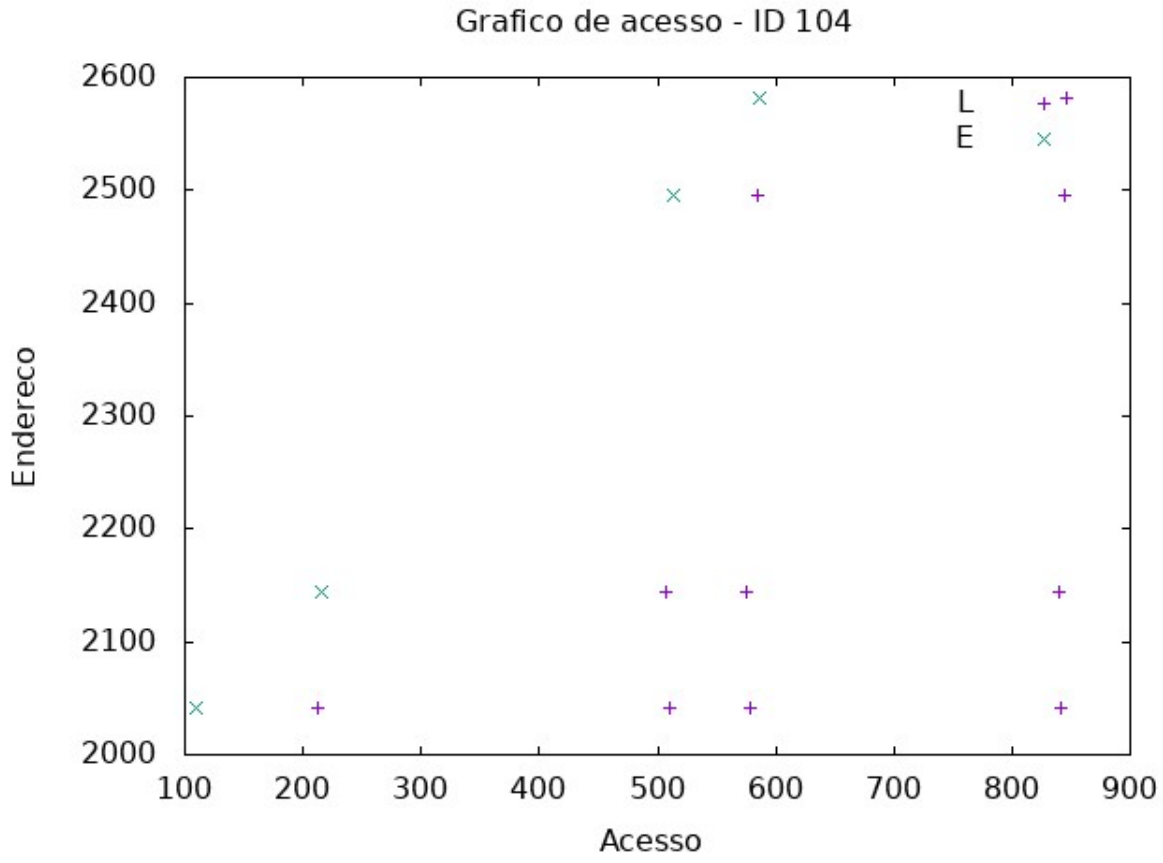


Figura 13 - Mapa de acesso para as URLs da Lista 4

Pelo mapa de acesso nota-se que para inserir uma nova URL, todas as presentes na Lista foram acessadas, seguindo o padrão visto para a Lista 3. Entretanto, pela análise do arquivo de entrada, observa-se que a segunda URL tem profundidade menor que a primeira. Sendo assim, o arquivo de entrada não precisa estar totalmente ordenado em ordem crescente para que todos os elementos sejam acessados, basta estar parcialmente ordenado, ou seja, a partir da segunda URL, todas as próximas têm profundidade maior ou igual a maior que já está na Lista. De fato, isso comprova o atendimento às especificações do projeto em que as primeiras URLs da Lista sejam as com menor profundidade ou, para uma mesma profundidade, aquelas conhecidas primeiro.

Para o último host conhecido do arquivo de entrada, as Figuras 14 e 15 mostram, respectivamente, o mapa de acesso para as células e para as URLs.

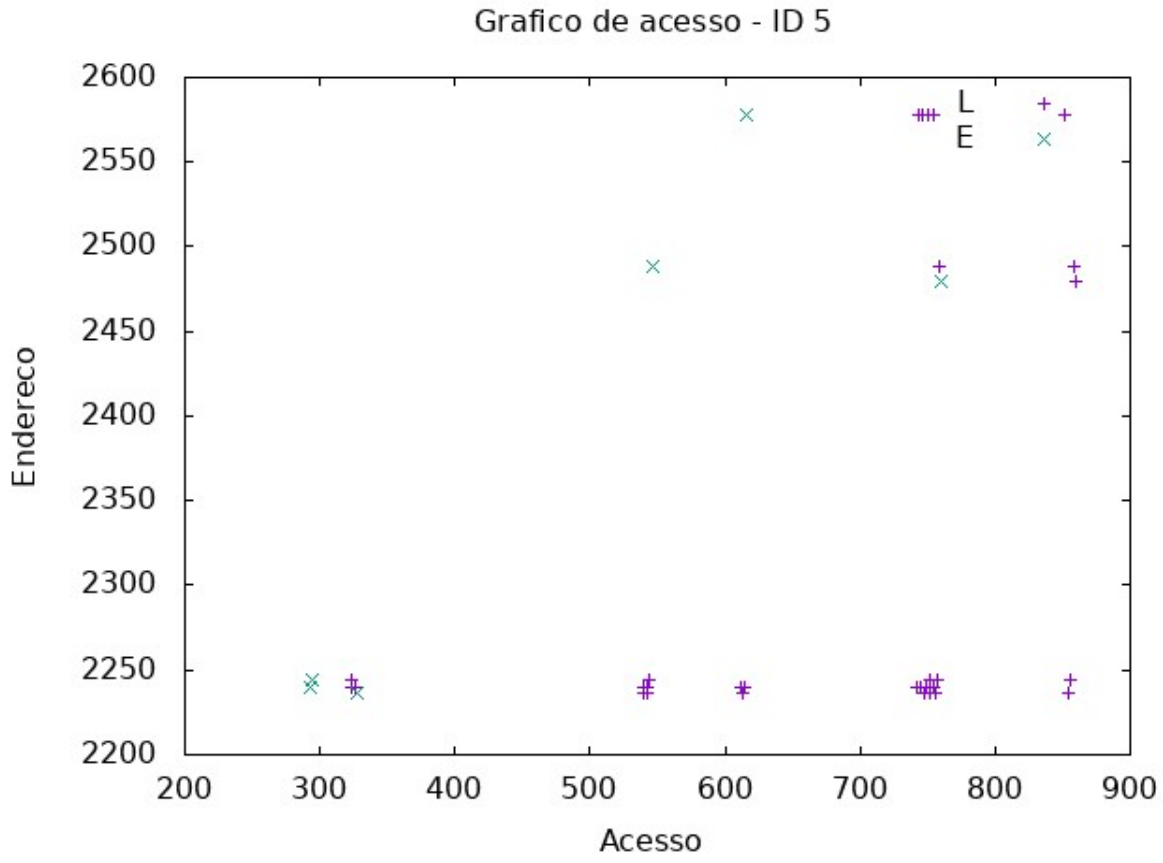


Figura 14 - Mapa de acesso para as células da Lista 5

Novamente observa-se a criação das células e a pesquisa prévia antes de inserir uma próxima.

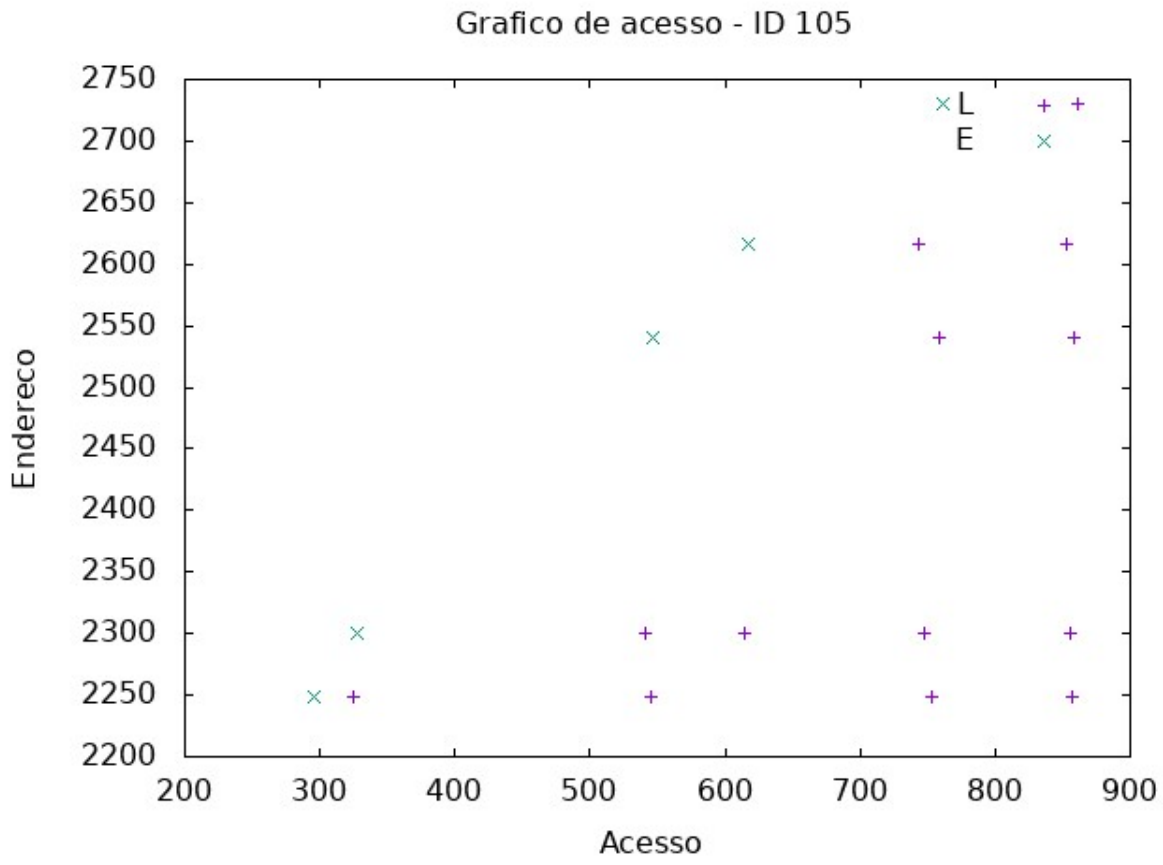


Figura 15 - Mapa de acesso para as URLs da Lista 5

Na Figura 15, observa-se também como os elementos são lidos para encontrar a posição correta para a inserção, permitindo a partir desse mapa, identificar a profundidade da URL a ser inserida em relação à profundidade das que já estão na Lista.

5.2.2. Distância de Pilha

A distância de pilha corresponde ao número de endereços de memória exclusivos acessados durante um período de reutilização, onde um período de reutilização é o tempo entre dois acessos sucessivos ao mesmo endereço de memória[3].

Para realizar a análise da distância de pilha, a execução dos vários comandos do Escalonador foi separado em fases, conforme a Tabela 3:

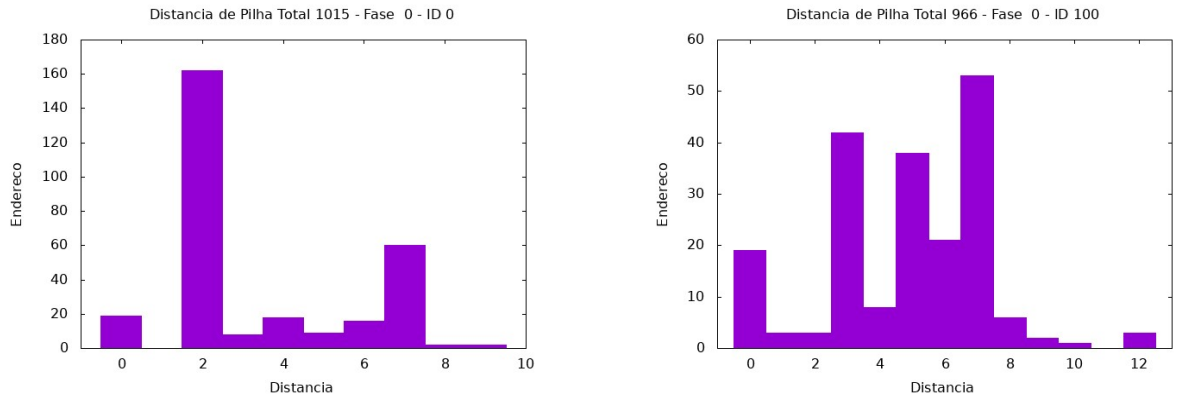
Tabela 3 - Fases de execução

Fase	Operação
0	ADD_URLS
1	ESCALONA_TUDO
2	ESCALONA

3	ESCALONA_HOST
4	VER_HOST
5	LISTA_HOSTS
6	LIMPA_HOST
7	LIMPA_TUDO

Como nestas análises só foram considerados os comandos ADD_URLS, ESCALONA_TUDO, LISTA_HOSTS e LIMPA_TUDO, têm-se apenas as Fases 0, 1, 5 e 7.

Na Figura 16 é apresentada a distância de pilha tanto para as células (a) quanto para os elementos (b) da Fila na Fase 0, ou seja, no processo de adicionar as URLs ao escalonador.

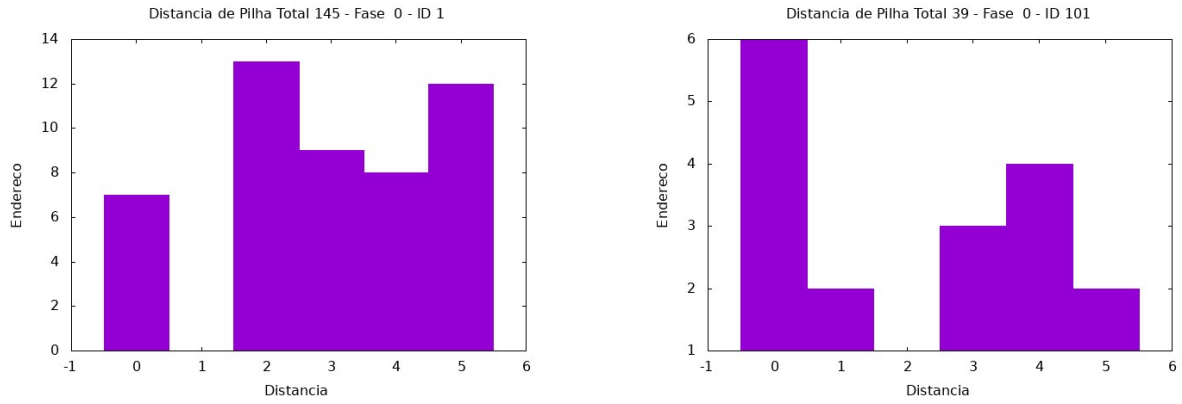


(a) (b)

Figura 16 - Distância de pilha para a Fila - (a) Células e (b) Elementos

Como foi observado também no mapa de acesso à memória, este processo é o mais custoso visto que são necessários vários acessos na Fila para verificar se o host já foi inserido. Além disso, como o método *PesquisaHost* utiliza as operações *pop_front* e *push_back* da Fila, então é de se esperar que a distância de pilha total para a Fila nesta Fase seja expressiva.

A distância de pilha para a primeira Lista é apresentada na Figura 17 (a) para as células e (b) para as URLs inseridas (Fase 0).

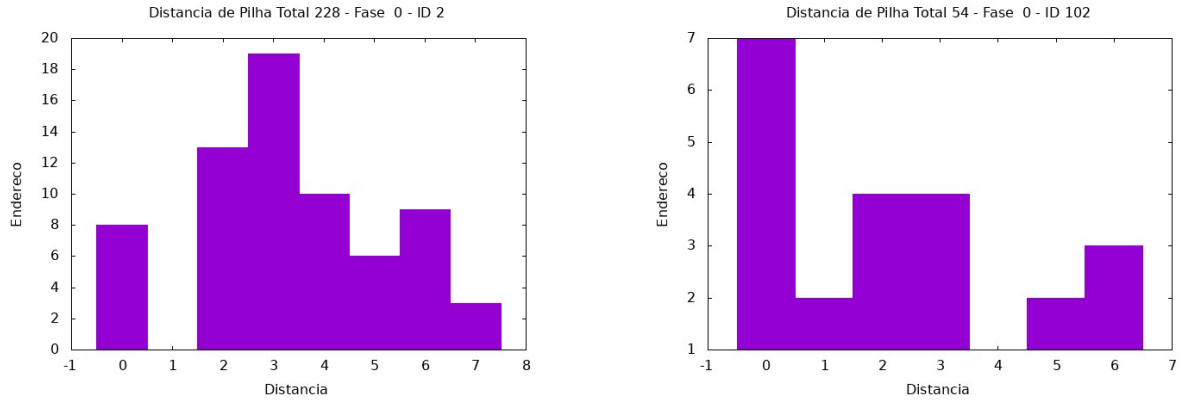


(a) (b)

Figura 17 - Distância de pilha para a Lista 1 - (a) Células e (b) Elementos

Pela análise da distância de pilha da primeira Lista, pode-se observar que a distância de pilha para as células é relativamente maior que para os elementos. Esse fato ocorre devido a estratégia adotada em utilizar ponteiros tanto para as células quanto para os elementos. Esta estratégia possibilita, na utilização dos métodos *get_item* e *insert_at*, percorrer na Lista acessando apenas os endereços das células até encontrar a posição a ser acessada e só então fazer o acesso ao elemento, fazendo com que a distância de pilha para os elementos seja relativamente inferior à distância de pilha para as células.

A Figura 18 mostra a distância de pilha para as células e elementos da segunda Lista na Fase 0.

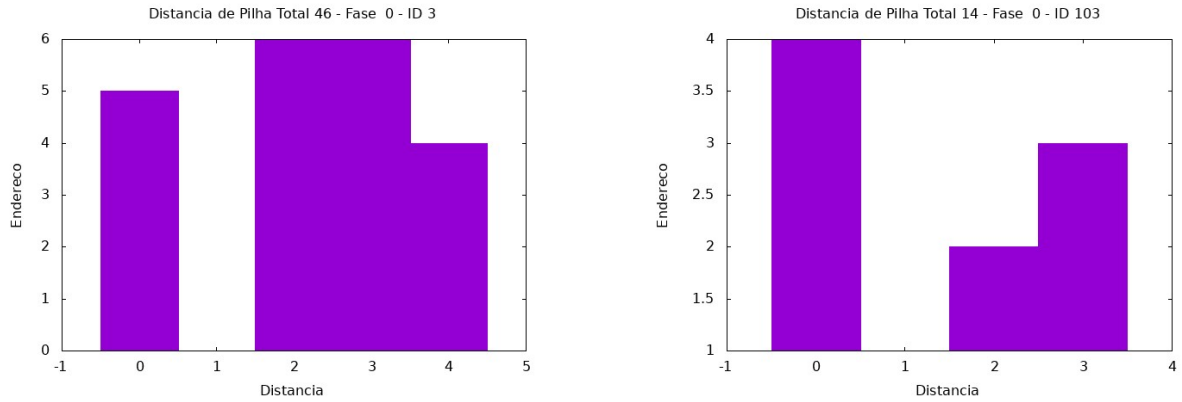


(a) (b)

Figura 18 - Distância de pilha para a Lista 2 - (a) Células e (b) Elementos

Novamente vemos o comportamento da estratégia adotada, gerando uma distância de pilha maior para as células. Além disso, pode-se observar que a distância de pilha para a Lista 2 é maior que para a Lista 1 devido ao fato de ser uma Lista com mais inserções, gerando uma maior movimentação na memória.

A Figura 19 mostra a distância de pilha na Fase 0 para a terceira Lista.

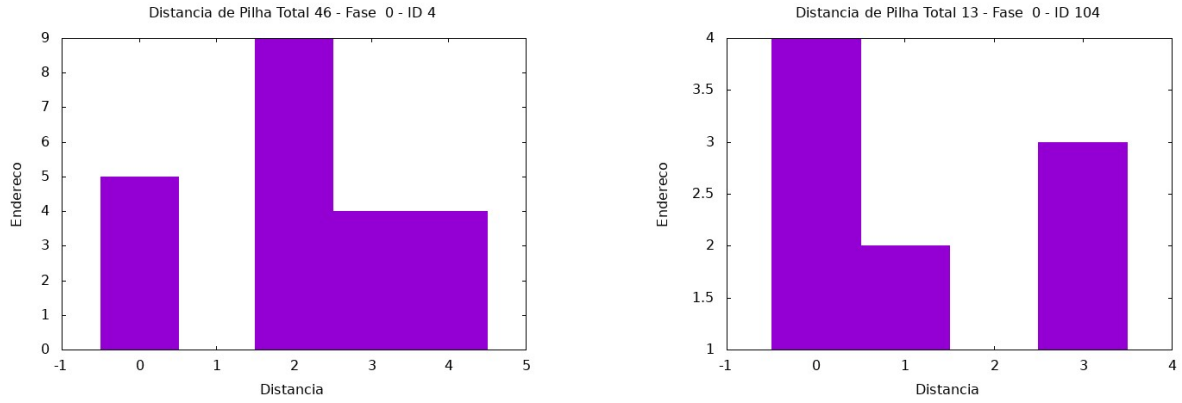


(a) (b)

Figura 19 - Distância de pilha para a Lista 3 - (a) Células e (b) Elementos

Como é uma Lista com uma quantidade menor de URLs em comparação com as duas primeiras Listas, a distância de pilha é menor.

A Figura 20 mostra a distância de pilha para a quarta Lista.

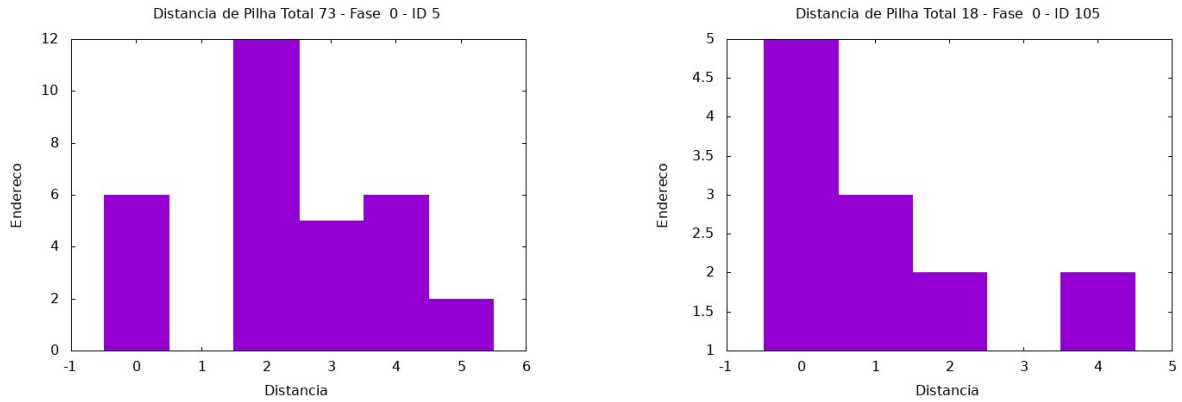


(a) (b)

Figura 20 - Distância de pilha para a Lista 4 - (a) Células e (b) Elementos

Observa-se que a distância de pilha para as Lista 3 e 4 são iguais para as células, porém com uma distribuição diferente e uma distância de pilha para os elementos menor na Lista 4. Isso permite concluir que para inserir as URLs do quarto host foi necessário uma menor manipulação de memória. Analisando a Tabela 1, é possível observar que, apesar de ter a mesma quantidade de URLs, as URLs do quarto host apresentam um espaçamento entre duas URLs sucessivas menor que do terceiro host, fazendo com que a memória da Lista 4 seja acessada em um menor intervalo de tempo e com isso gerando uma distância de pilha menor.

A Figura 21 mostra a distância de pilha para a quinta Lista.



(a) (b)

Figura 21 - Distância de pilha para a Lista 5 - (a) Células e (b) Elementos

Observa-se, pela Figura 21, que a Lista 5 armazena mais URLs que as duas anteriores gerando uma distância de pilha maior tanto para as células quanto para os elementos.

Na Fase 1 (ESCALONA_TUDO), como as URLs são apenas retiradas das Listas uma a uma, iniciando na primeira até a última (Lista ordenada) a distância de pilha para as Listas é zero. Como na Fila as células são retiradas e depois inseridas novamente para manter os hosts conhecidos, tem-se a distância de pilha apresentada na Figura 22.

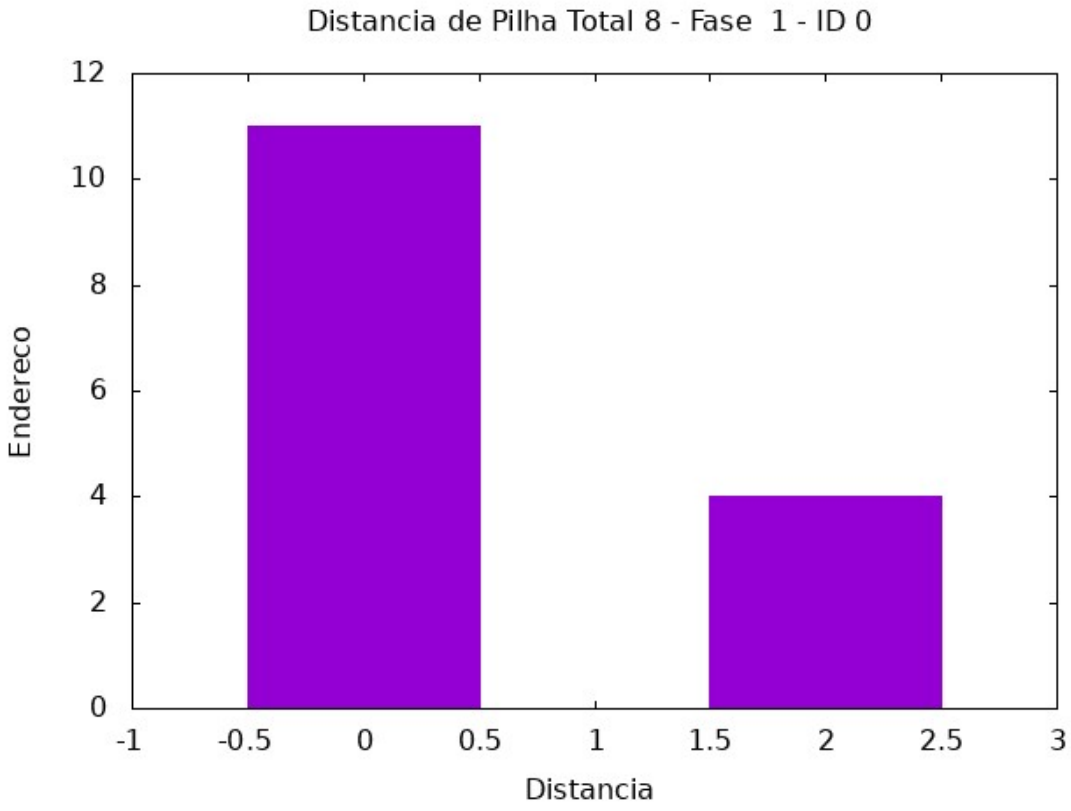
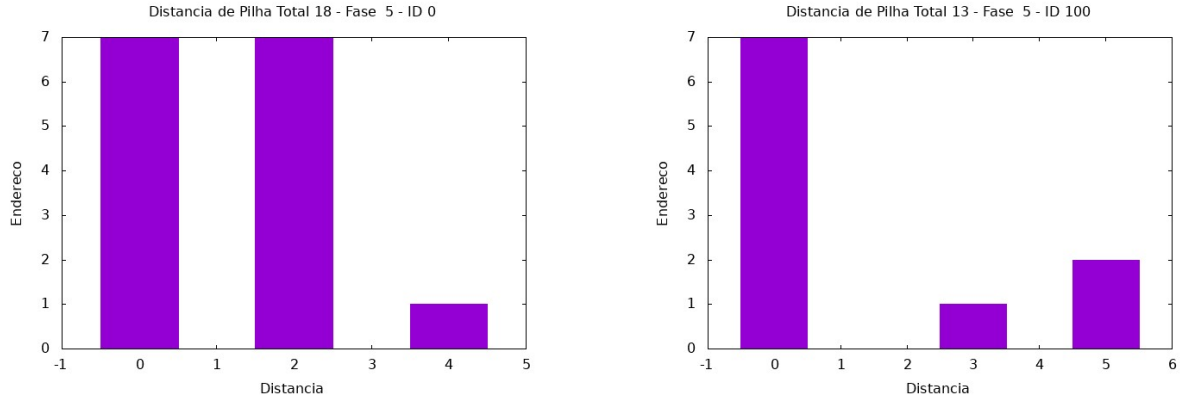


Figura 22 - Distância de pilha para as células da Fila

Na Fase 5 (LISTA_HOSTS), tem-se o acesso às células da Fila e aos elementos para listar os host, de forma que a Figura 23 mostra a distância de pilha para (a) células e (b) elementos da Fila Encadeada.



(a) (b)

Figura 23 - Distância de pilha na Fase 5 para a Fila - (a) Células e (b) Elementos

Como na Fase 7 (LIMPA_TUDO) tem-se apenas o acesso às células da Fila para fazer a desalocação de cada célula, tem-se a distância de pilha mostrada na Figura 24, com uma distância de pilha total de 4.

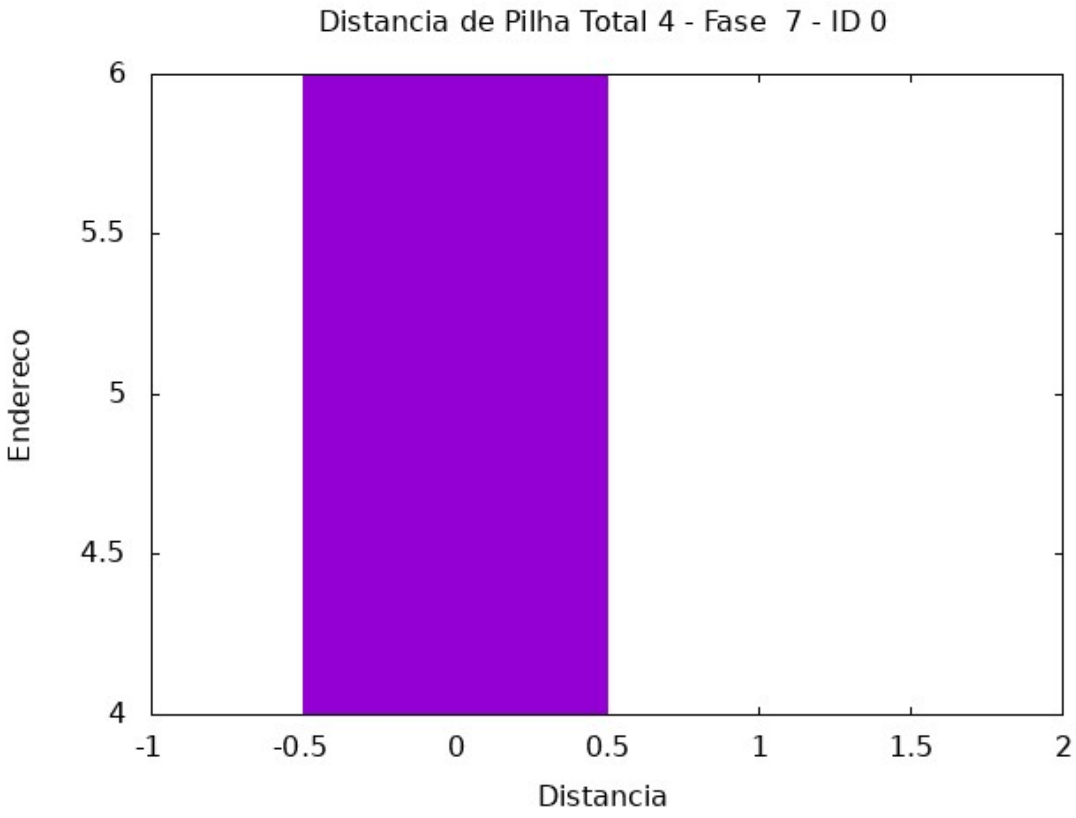


Figura 24 - Distância de pilha da Fila na Fase 7

6. Conclusão

Este trabalho apresentou o desenvolvimento de um escalonador de URLs utilizando a estratégia depth-first para escalonar as URLs conhecidas. O projeto do escalonador envolveu a utilização dos TADs Fila Encadeada e Lista Encadeada, além de TAD auxiliar para tratar os casos de URLs inválidas.

O programa foi analisado quanto a complexidade de tempo e de espaço e testes de desempenho e de localidade de referência foram realizados para avaliar o comportamento global dos códigos desenvolvidos. A análise de complexidade de tempo do escalonador foi realizada dividindo o problema em três cenários possíveis, porém os testes foram realizados para o cenário mais geral.

Para um trabalho futuro, seria interessante realizar outros testes para os demais cenários afim de validar todas as análises feitas.

Bibliografia

[1] Pappa, Gisele L., Meira Jr., Wagner. Slides virtuais da disciplina de Estrutura de Dados 2021/2. Disponibilizado via Moodle. DCC. Universidade Federal de Minas Gerais

[2] William., Stallings (2010). Computer organization and architecture: designing for performance (8th ed.). Upper Saddle River, NJ: Prentice Hall. ISBN 9780136073734. OCLC 268788976

[3] Kecheng Ji, Ming Ling, Longxing Shi. Using the first-level cache stack distance histograms to predict multi-level LRU cache misses. Microprocessors and Microsystems, v. 55, 2017, p55-69, ISSN 0141-9331, https://doi.org/10.1016/j.micpro.2017.10.001.

Apêndice A

Instruções de compilação e execução

A.1 Compilação

O programa pode ser compilado utilizando o Makefile presente na raiz do projeto, utilizando os seguintes comandos:

make all - compila todo o projeto gerando o executável ***main*** na pasta bin;

make clean - exclui os códigos objetos do diretório obj e o executável da pasta bin;

A.2 Execução

Depois de compilado, o programa é executado pela linha de comando com o seguinte comando:

Dentro da raiz do projeto: **./bin/main** <*arquivo*> **-b**

onde: <*arquivo*> é o nome do arquivo que será lido com os comandos a serem executados pelo escalonador;

A opção **-b** é opcional e representa a escolha da estratégia ***breadth-fisrt*** de escalonamento.

Registro de acesso

Para fazer o registro de acesso à memória é necessário acrescentar a flag **-l** no comando.

Apêndice B

Arquivo utilizado para os testes de Localidade de Referência

ADD_URLS 26
http://mqb.darz.com/bljpt/nfw/fjm.html
http://mqb.darz.com/rws/sbcnuv.html
http://mqb.darz.com/saqx/qcaceh/zvfrkml.html
http://uewhs.com/pqpx/xkit/xacbh/icqc.html
http://uewhs.com/omfg/dwf/pxiqvku/dlcgdew.html
http://xpklorell.papq.kho.mco.com/ohordtq/wcsgspq/msboagu.html
http://xpklorell.papq.kho.mco.com/xnzlgdg/btrwbln/deuguu.html
http://qcl.ivsw.kqtbxix.com/rubetok/hoachwdv/xrdryxl/dqtukw.html
http://mqb.darz.com/juu/cibxub/enmeyat/mydi/xloghiq.html
http://mqb.darz.com/vih/uvsu/ypayulyei/otehzrii.html
http://qcl.ivsw.kqtbxix.com/ggkbbipz/zu.html
http://uewhs.com/udfykgru/zgiooobp.html
http://uewhs.com/wphapj/dqhdc/wdtxj.html
http://iddqscdxrj.wfrxsjy.dbef.rcbyne.com//aux/pus/hiixqm.html
http://iddqscdxrj.wfrxsjy.dbef.rcbyne.com/cvud/uyiby/mwsiqyo.html
http://uewhs.com/mzev/zvjegeb/cfuftsx/xtig/eehkchzdfl.html
http://uewhs.com/fnxzt/svbspk.html
http://xpklorell.papq.kho.mco.com/bppkqtp/buot/qcwi.html
http://mqb.darz.com/ujjd/tge.html
http://qcl.ivsw.kqtbxix.com/aijvw/aubwewpj/gehlj/pbp.html
http://iddqscdxrj.wfrxsjy.dbef.rcbyne.com/dzubdubz/fsp/qwuzifwov/dwyvvbur/mgyjgfdx/nun
.html
http://qcl.ivsw.kqtbxix.com/plw/upfxlz/nhkw/anlt.html
http://iddqscdxrj.wfrxsjy.dbef.rcbyne.com/cdds/oyvegur.html
http://xpklorell.papq.kho.mco.com/moxeqm/owrg/lk/meahkgccn.html
http://uewhs.com/veym/xhlrnunyf/rhbas/uyg/oubutpni/wfjqsjx.html
http://iddqscdxrj.wfrxsjy.dbef.rcbyne.com/rxx/wctd/eogvbpkx/gdir/cr/ifpgyn.html
LISTA_HOSTS
ESCALONA_TUDO
LIMPA_TUDO