Trabalho Prático 1 de Estrutura de Dados

Artur Gaspar da Silva, 2020006388

Introdução

Este arquivo visa documentar e analisar o funcionamento do programa “escalonador”, como primeiro trabalho prático da disciplina de Estrutura de Dados do curso de Bacharelado em Ciência da Computação da Universidade Federal de Minas Gerais (UFMG). Buscamos ordenar e imprimir URLs, sendo que eles e comandos são lidos em um arquivo especificado como argumento da linha de comando e salvando os resultados em outro arquivo, com o mesmo nome do arquivo de entrada concatenado a um “-out”. Implementamos três versões: a requisitada e dois desafios extras.

Na versão principal, os URLs são ordenados por profundidade e host, de forma que um host inserido antes é priorizado (veja os desafios para outras situações), e no mesmo host são priorizados URLs com profundidade menor. A profundidade de um URL é definida como a quantidade de barras no *path* do URL. Incluímos também uma análise da complexidade do programa em diversos casos, e uma análise experimental do desempenho do mesmo.

Método

Implementamos seis tipos de dados distintos: “URL” e “URL_Node”, “Host”, “Host_Node”, “Fila_Hosts”, “Escalonador” e “comando”. Temos também métodos internos de cada tipo de dados, duas macros, uma função para obter o nome do arquivo de saída do programa e outra para obter o nome do arquivo de registro, uma função para explicar ao usuário o uso do programa e a função principal do programa. Além disso, o projeto inclui um Makefile para compilar o programa, e a divisão de arquivos em pastas “src”, com as implementações dos tipos de dados e funções, “include” com os arquivos de cabeçalho que definem os tipos de dados como “class”, além de algumas macros. Por fim, a pasta “obj” guarda os arquivos de objeto compilados e a pasta “bin” guarda o arquivo executável resultante do programa.

Estruturação dos arquivos de entrada e saída:

O arquivo de entrada contém a saída dos comandos requisitados na entrada, e o arquivo de entrada contém algum dos seguintes comandos:

- 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 de ordenamento especificadas na seção seguinte. Quando escalonadas, as URLs são exibidas e removidas da lista.
- ESCALONA <quantidade>: escalona no máximo <quantidade> URLs, removendo-as em seguida.
- ESCALONA_HOST <host> <quantidade>: são escalonadas <quantidade> URLs deste host, e as remove em seguida.
- 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, mas não remove o host.
- LIMPA_TUDO: limpa todas as URLs e hosts.

Regras para as URLs:

- Apenas URLs de protocolo *http* são permitidas, logo “https://www.google.com” seria ignorado.
- A barra "/" ao final da URL não deve ser considerada, logo "http://ufmg.br/" deve ser salva como "http://www.ufmg.br".
- O "www." no início da URL não deve ser considerado, logo "http://www.ufmg.br" será salva como "http://ufmg.br"

- Apenas páginas HTML são permitidas. Para isto, arquivos com as seguintes extensões serão descartados: .jpg, .gif, .mp3, .avi, .doc e .pdf, logo “https://www.ufmg.br/ceu/assets/site/doc/Regulamento_Interno_CEU.pdf” será ignorada.
- Desconsiderar #<fragmento> nas URLs, logo “https://ufmg.br/#acesso-busca” será salva como “https://ufmg.br”.

Ordenamento das URLs:

- URLs de hosts adicionadas primeiro serão escalonadas primeiro.
- Caso duas URLs possuam mesmo host, priorizamos URLs com menor profundidade. *Profundidade* é definida como a quantidade de barras no *path* do URL.
- Caso duas URLs tenham mesma profundidade e mesmo host, priorizamos aquelas que foram adicionadas primeiro.

Atributos e métodos do tipo “Comando”:

O tipo “Comando” é especificado no arquivo de cabeçalho “comando.h”, possuindo cinco atributos privados:

- *id_comando*: Um inteiro que identifica o tipo de comando em questão da seguinte forma:
  0. ADD_URLS
  1. ESCALONA_TUDO
  2. ESCALONA
  3. ESCALONA_HOST
  4. VER_HOST
  5. LISTA_HOSTS
  6. LIMPA_HOST
  7. LIMPA_TUDO
- *host* especifica o host passado como argumento, se aplicável.
- *urls* é uma lista dos urls passados como argumento, utilizado somente para o comando “ADD_URLS”.
- *quantidade* especifica a quantidade passada como argumento, se aplicável
- *atual_url_pos* especifica a próxima posição da lista *urls* que ainda não foi lida, se aplicável.

No arquivo “comando.cpp” implementamos os métodos públicos usados para manipular o tipo “Comando”:

- *get_id* não recebe argumento e retorna o inteiro identificador do tipo de comando.
- *get_host* não recebe argumento e, caso o comando seja de um tipo que recebe um host como argumento, retorna a string correspondente ao host, senão o programa termina e imprime uma mensagem de erro.
- *more_urls* não recebe argumento e retorna um valor booleano verdadeiro se ainda há URLs a serem lidos, senão retorna falso. Caso o comando não seja do tipo “ADD_URLS”, senão o programa termina e imprime uma mensagem de erro.
- *get_url* também não recebe argumento e retorna o próximo url a ser lido. Caso o comando não seja do tipo “ADD_URLS”, senão o programa termina e imprime uma mensagem de erro. O program termina e imprime mensagem de erro se não há URLs a serem lidos.
- *get_quantidade* não recebe argumento, caso o comando seja de um tipo que recebe uma quantidade como argumento, retorna essa quantidade, senão o programa termina e imprime uma mensagem de erro.
- *~Comando* é o destrutor do comando, que não recebe argumento e libera a memória alocada dinamicamente para armazenar os urls, se aplicável.

Por fim, no mesmo arquivo, implementamos também o operador “>>” para ler um comando a partir de uma stream:

- *operator>>* define o operador de leitura de stream para o tipo “Comando”, que efetivamente constrói um comando a partir do que for lido na stream.

Atributos e métodos do tipo “Escalonador”:

O tipo “Escalonador” é especificado no arquivo de cabeçalho “escalonador.h”, possuindo dois atributos privados:

- *arq_saida* especifica a stream do arquivo em que deve ser colocado as saídas dos comandos do programa.
- *fila* especifica a fila dos hosts dos urls já lidos (do tipo *Fila_Hosts).*

No arquivo “escalonador.cpp” implementamos os métodos públicos usados para manipular o tipo “Escalonador”:

- *Escalonador* define o construtor da classe, que define a stream do arquivo de saída a ser utilizado a partir do caminho para o mesmo.
- *executar_comando* executa o comando passado como argumento (do tipo *Comando*), não retornando nada.
- *~Escalonador* define o destrutor do escalonador, que fecha o arquivo de saída e limpa todos os urls e hosts utilizados.

No mesmo arquivo implementamos os métodos privados utilizados internamente pela classe “Escalona”:

- *add_url* recebe como argumento uma string que representa o url a ser adicionado, e adiciona o tipo *URL* correspondente à lista de urls do host, de forma a manter a lista ordenada.
- *escalona_tudo* escalona todos os urls recebidos, imprimindo-os em ordem e deletando-os em seguida. Implementamos isso escalonando até 2147483647 URLs, que na maioria dos usos será mais que o suficiente, já que cada URL tem pelo menos 7 bytes ("http://"), passar esse limite exigiria mais que 14Gb na memória secundária.
- *escalona* recebe como argumento a quantidade de urls a ser escalonada, e imprime no máximo essa quantidade de urls em ordem, deletando-os em seguida. O método retorna a quantidade de urls efetivamente escalonados (caso a quantidade recebida seja maior que a quantidade de urls efetivamente guardados, somente todos presentes são escalonados).
- *escalona_host* recebe como argumento uma string representando o host a ser considerado, e uma quantidade. O método imprime até no máximo a quantidade de urls especificados e os deleta em seguida. O retorno é a quantidade de urls efetivamente escalonados (caso a quantidade recebida seja maior que a quantidade de urls guardados, todos presentes são escalonados).
- *escalona_host_interno* é um método auxiliar que recebe como argumento o nó do host do qual se deseja escalonar e a quantidade máxima a escalonar. O retorno do método é a quantidade de urls efetivamente escalonadas.
- *ver_host* recebe como argumento a string correspondente ao host que se deseja inspecionar, e imprime todos os urls daquele host em ordem, mas sem deletá-los.
- *lista_hosts* não recebe nenhum parâmetro e lista todos os hosts guardados em ordem.
- *limpa_host* recebe como argumento a string correspondente ao host que se deseja limpar, e deleta todos os URLs daquele host, mas sem deletar o host da lista.
- *limpa_tudo* não recebe nenhum parâmetro, e remove todos os hosts e urls da fila.

Atributos e métodos do tipo “Host_Node”:

O tipo “Host_Node” é especificado no arquivo de cabeçalho “fila_hosts.h”, possuindo dois atributos públicos:

- *host* é o host associado a esse nó, do tipo *Host*.
- *próximo* é o apontador para o próximo nó de host, do tipo *Host_Node*.

No arquivo “ fila_hosts .cpp” implementamos o construtor e destrutor públicos usados para criar e destruir o tipo “Host_Node”:

- *Host_Node* é o construtor do tipo *Host_Node*, que recebe um argumento do tipo *Host* que define o host correspondente a esse nó.

- *~Host_Node* é o destrutor do tipo *Host_node*, que não recebe nenhum parâmetro e limpa todos os urls do host associado.

Atributos e métodos do tipo "Fila_Hosts":

O tipo "Fila_Hosts" é especificado no arquivo de cabeçalho "fila_hosts.h", possuindo dois atributos privados:

- *no_frontal* é um apontador para o tipo *Host_Node*, e representa o nó do primeiro host na fila.
- *tamanho* é um inteiro que representa a quantidade de hosts presentes na fila.

No arquivo "fila_hosts .cpp" implementamos os métodos públicos usados para manipular o tipo "Fila_Hosts":

- *Fila_Hosts* é o construtor da classe, que não recebe parâmetros e inicializa o ponteiro inicial como vazio e o tamanho como nulo.
- *add_host* adiciona um host passado como argumento à fila.
- *get_front_host* não recebe parâmetros e retorna um ponteiro pro nó do host inicial.
- *remove_front_host* não recebe parâmetros e não tem retorno, e deleta o primeiro host da fila.
- *get_host* recebe como parâmetro a string correspondente ao host, e retorna um ponteiro pro nó deste, caso presente na fila. Caso não esteja presente na fila, retorna um ponteiro nulo.
- *remove_host* recebe como parâmetro a string corresponde ao url que se deseja remover, e o remove caso esteja presente. Caso não esteja presente, nada é feito, e o método não tem retorno.
- *get_tamanho* não recebe parâmetros e retorna um inteiro que representa a quantidade de hosts atualmente presentes na fila.
- *vazia* não recebe parâmetros e retorna um valor booleano verdadeiro se a fila está vazia, falso se possui pelo menos um host.
- *clear* remove todos os hosts da fila.
- *~Fila_Hosts* é o destrutor da classe, deleta todos os hosts da fila, efetivamente chamando o método *clear*.

Atributos e métodos do tipo "URL_Node":

O tipo "URL_Node" é especificado no arquivo de cabeçalho "host.h", possuindo dois atributos privados:

- *url* é do tipo *URL* e especifica o url correspondente ao nó.
- *proximo* é um ponteiro para o tipo *URL_Node*, que corresponde ao nó do próximo url.

No arquivo " host .cpp" implementamos o construtor público usado para criar o tipo "URL_Node":

- *URL_Node* é o construtor do tipo, que recebe como argumento o url correspondente ao nó, e o coloca em *próximo*.

Atributos e métodos do tipo "Host":

O tipo "Host" é especificado no arquivo de cabeçalho "host.h", possuindo três atributos privados:

- *host_string* é a string que representa o host em questão.
- *no_frontal* é um ponteiro para o tipo *URL_Node* que representa o primeiro host da fila.
- *tamanho* é um inteiro que identifica quantos urls esse host possui no momento.

No arquivo "host.cpp" implementamos os métodos públicos usados para manipular o tipo "Host":

- *Host* é o construtor do tipo, inicializando o tamanho como zero, a string do host e o nó frontal como ponteiro nulo.
- *base_string* retorna a string do host e não recebe argumentos.
- *add_url* adiciona, já de forma ordenada, um url ao host, que é passado como argumento. Não há retorno.
- *get_tamanho* retorna a quantidade de URLs atualmente associadas ao host em questão.

- *get_first_url* retorna o nó do primeiro url desse host, e *nullptr* se não há urls. Não há parâmetro,
- *remove_first_url* não recebe parâmetros e não tem retorno, mas remove o primeiro url da fila.
- *vazio* não recebe parâmetros e retorna um valor booleano verdadeiro se o host não tem nenhum url associado, e falso se possui.
- *limpar* não recebe parâmetros nem tem retorno, mas remove todos os urls do host.
- *~Host()* é o destrutor da classe, deleta todos os URLs da fila, efetivamente chamando o método *limpar*.

Atributos e métodos do tipo "URL":

O tipo "URL" é especificado no arquivo de cabeçalho "url.h", possuindo um atributo privado:

- *url_string* é onde ficará armazenada a string da url, em sua forma tratada.

No arquivo "url.cpp" implementamos os métodos públicos usados para manipular o tipo "URL":

- *URL* é o construtor do tipo, que recebe a uma string a partir da qual se deseja obter o url, trata a sring se for válida e depois armazena em *url_string*.
- *as_string* retorna a url tratada em sua forma de string. Não recebe parâmetros.
- *get_host_string* retorna a string correspondente ao host da url. Não recebe parâmetros.

Também no arquivo "url.cpp" implementamos métodos privados usados internamente pelo tipo "URL":

- *url_valido* recebe como parâmetro uma string e retorna um valor booleano verdadeiro se a string é válida segundo as regras especificadas na seção "Método" dessa documentação, e falso caso contrário.
- *remove_fragmento* recebe como parâmetro uma string de url e retorna uma versão da string que não possui fragmento (remove tudo depois do último "#", incluindo o hashtag).
- *extensao_valida* recebe como parâmetro uma string que já teve possível fragmento removido, e retorna um valor booleano falso se a string terminar com extensão inválida (.jpg, .gif, .mp3, .avi, .doc e .pdf), e verdadeiro caso contrário.
- *remove_www* recebe como parâmetro uma string e retorna uma versão da string uma versão que não possui "www.".
- *url_tratado* recebe como parâmetro uma string de url e retorna a versão tratada do mesmo. Isso inclui remover o www e remover o fragmento e a barra no final.

Por fim, no mesmo arquivo, implementamos também o operador "<<" para escrever o url em questão em uma stream, e o operador ">=" usado na ordenação das urls:

- *operator<<* define o operador de escrita em stream a partir do tipo url, sendo escrita a versão tratada do mesmo.
- *operator>=* define o operador de comparação entre urls de mesmo host, utilizado no ordenamento dos mesmos.

Tipo "memlog" e funções para a sua implementação:

O tipo *memlog_tipo* é especificado no arquivo *memlog.h*, possuindo 7 atributos:

- *log* define o arquivo de registro do desempenho e padrão de acesso à memória do programa.
- *clk_id* define o modo do relógio utilizado, segundo a especificação biblioteca *time.h*.
- *inittime* registra o tempo inicial de execução do programa no formato especificado na biblioteca *time.h*.
- *count* é um cantador de quantas operações de registro foram realizadas.
- *regmem* informa se o padrão de acesso à memória deve ser registrado, ou só o tempo de execução total.
- *fase* identifica em qual fase de registro o programa está. Atualmente o programa só possui uma fase de registro, este atributo facilita adaptações futuras do registro.

- *ativo* identifica se o padrão de acesso à memória e o desempenho do programa estão sendo registrados atualmente ou não.

Note que possuímos uma variável estática *ml* de registro de acessos em *memlog.cpp*, e algumas das funções implementadas fazem referência a ele. Além disso possuímos a função auxiliar *clkDifMemLog* que calcula a diferença entre dois instantes de tempo passados como parâmetros e salva num terceiro parâmetro passado por referência. Por fim, temos as macros *MLATIVO* e *MLINATIVO* usadas para identificar se o registro está ativo ou não.

Também no arquivo *memlog.h* especificamos as funções para registrar os padrões de acesso à memória e o desempenho computacional:

- *iniciaMemLog* recebe como parâmetro o nome do arquivo onde serão armazenados os registros de desempenho e padrão de acesso à memória e inicializa a variável estática *ml* do tipo *memlog_tipo*.
- *ativaMemLog* atualiza o estado do *memlog_tipo* estático para ativado.
- *desativaMemLog* atualiza o estado do *memlog_tipo* estático para desativado.
- *defineFaseMemLog* recebe um inteiro como parâmetro que representa a fase do registro, e armazena no atributo *fase* da variável estática do tipo *memlog_tipo*.
- *leMemLog* recebe como parâmetro uma posição de memória lida e um tamanho que representa a quantidade de bytes lidos, e armazena essas informações no arquivo de registro, juntamente com informações sobre o tempo em que ocorreu a leitura.
- *escreveMemLog* recebe como parâmetro uma posição de memória escrita e um tamanho que representa a quantidade de bytes escritos, e armazena essas informações no arquivo de registro, juntamente com informações sobre o tempo em que ocorreu a leitura.
- *finalizaMemLog* conclui o registro, marcadno a variável estática *ml* do tipo *memlog_tipo* como desativada, além de colocar o registro final no arquivo de fechá-lo.

Temos também uma função auxiliar.

Macros de msgassert.h:

Possuímos quatro macros neste arquivo para auxiliar o tratamento de erros:

- *avisoAssert* recebe um valor booleano e uma mensagem, e caso o valor seja falso ele chama a macro *__avisoassert.*
- *erroAssert* recebe um valor booleano e uma mensagem, e caso o valor seja falso ele chama a macro *__avisoassert*.
- *__avisoassert* é uma macro auxiliar utilizada pelo *avisoAssert* que imprime a mensagem de aviso no *stderr* e não interrompe a executção do programa.
- *__erroassert* é uma macro auxiliar utilizada pelo *avisoAssert* que imprime a mensagem de aviso no *stderr* e interrompe a executção do programa.

Arquivo principal “escalonar.cpp”:

São implementadas três funções nesse arquivo:

- *uso* é uma função que não recebe nenhum argumento e imprime no *standard output* do programa uma mensagem de ajuda sobre quais argumentos são aceitos pelo programa.
- *get_nome_saida* é uma função que recebe como argumento o nome do arquivo de entrada e retorna o nome do arquivo de saída correspondente (ou seja, com -out concatenado ao nome mas não à extensão: “teste.txt” se torna “teste-out.txt”).
- *get_nome_registro* é uma função que recebe como argumento o nome do arquivo de entrada e retorna o nome do arquivo de registro correspondente (ou seja, com -log concatenado ao nome mas não à extensão: “teste.txt” se torna “teste-log.txt”).
- *main* é a função principal do programa, que recebe os argumentos de linha de passados e faz as chamadas para as funções do programa adequadas para identificar o arquivo de entrada, executar seus comandos e colocar os resultados no arquivo de saída.

Análise de Complexidade

As principais tarefas realizadas pelo programa são: identificação de comandos passados no arquivo de entrada, que envolve adição, ordenamento e impressão de hosts e URLs recebidos.

Abaixo faremos a análise de complexidade assintótica do programa nos principais casos possíveis na ordem em que são executados:

- Primeiramente identificamos o arquivo de entrada, o nome do arquivo de saida correspondente e instanciamos tipos Comando e Escalonador que serão utilizados durante a execução do programa. As etapas que envolvem o nome do arquivo de entrada ou saída tomam tempo **Θ((e)**, sendo **e** o tamanho do nome do arquivo de entrada. Isso ocorre porque copiamos o nome de entrada para podermos ustilizar o tipo “std::string” no lugar de “char*”.
- Após o processamento dos argumentos, instanciação das streams de leitura e escrita e dos tipos Escalonador e Comando, temos o início do registro de padrão de acesso à memória com *iniciamemlog* e sua ativação com *ativamemlog*. O segundo procedimento simplesmente marca *ml* como ativa, sendo claramente sempre executado em tempo constante e com complexidade de espaço constante em relação a qualquer parâmetro cabível. O primeiro procedimento inicializa a variável estática *ml* e abre o arquivo passado como argumento, portanto podemos dizer que sua complexidade assintótica é **Θ(1)** em relação ao seu parâmetros de entrada: qualquer que seja o nome do arquivo de entrada, o tempo de execução se mantém constante. A complexidade de espaço de ambas também é **Θ(1)**, pois independe de seus parâmetros.

A próxima etapa de execução do programa depende não só da quantidade, mas também de quais são os comandos lidos no arquivo de entrada. O tempo de execução total será a soma do tempo de leitura e execução de cada um dos comandos no arquivo. Assintoticamente, será **Θ(** $\sum_{i=1}^{c} (l_i + e_i)$ **)**, sendo **e_i** o tempo de execução do i-ésimo comando, **l_i** o tempo de leitura do mesmo, e **c** a quantidade de comandos. O registro de padrão de acesso e localidade influencia no tempo de execução mas essa influência não se observa assintoticamente.

Agora analisaremos o fluxo do programa de acordo com cada comando:

- ADD_URLS <quantidade> tem tempo de leitura e execução **O(n k s)** se **n** é a quantidade de URLs lidos, **s** o tamanho do maior *URL*, e **k** a quantidade de URLs atualmente presentes, pois cada url deve ser lido individualmente da memória secundária, e temos que copiar cada caractere para a string correspondente na memória principal, passando por possivelmente todos os outros. A complexidade de espaço é **O(n s)**, pois é a quantidade de caracteres que será copiada.
- ESCALONA_TUDO tem tempo de leitura **O(1)** (pois não possui argumento) e tempo de execução **O(m s)** se **m** é a quantidade de URLs salvas no momento e **s** é o tamanho do maior *URL*. A complexidade de espaço será **O(1)** em ambos pois a memória extra necessária para sua execução é constante.
- ESCALONA <quantidade> tem tempo de leitura **O(log(q))** se **q** é a quantidade informada, pois é um número inteiro representado por **log(q)** caracteres. O tempo de execução é **O(q s)** se **s** é o tamanho do maior *URL*, pois teremos que imprimir exatamente **q** urls, de tamanho no máximo **s**. A complexidade de espaço é **O(1)** tanto na leitura quanto na execução, pois o espaço extra necessário para ambas independe dos parâmetros de entrada e do estado atual dos TADs.
- ESCALONA_HOST <host> <quantidade> tem tempo de leitura **O(h log(q))** se **h** é o tamanho do argumento host e **q** o valor de quantidade informado, pois representa a quantidade de caracteres lidas dos parâmetros do comandos. O tempo de execução será **O(q s)**, com **q** sendo a quantidade passada e **s** sendo o tamanho do maior URL, no pior caso (quando o host tem **q** ou mais URLs salvos), e **O(1)** no melhor caso, que seria justamente quando o host não tem nenhum URL salvo. A complexidade de espaço da leitura e da execução será **O(1)**, pois o espaço extra necessário para a ambas independe dos parâmetros de entrada e do estado atual dos TADs.

- VER_HOST <host> tem tempo de leitura **O(h)**, com **h** sendo o tamanho do argumento host, pois representa a quantidade de caracteres na leitura dos parâmetros do comando. O tempo de execução será **O(t s)**, em que **t** é o tamanho do host, ou seja, a quantidade de URLs que ele possui, e **s** é o tamanho (em caracteres) do maior URL desse host. A complexudade de espaço da leitura e da execução será **O(1)**, pois o espaço extra necessário para ambas independe dos parâmetros de entrada e do estado atual dos TADs.
- LISTA_HOSTS tem tempo de leitura **O(1)** por não possuir argumentos, e tempo de execução **O(f t)** com **f** sendo o tamanho da fila de hosts, ou seja, a quantidade de hosts atualmente salvas, e **t** o tamanho (em caracteres) do maior host. A complexidade de espaço da leitura e da execução será **O(1)**, pois o espaço extra necessário para ambas independe dos parâmetros de entrada e do estado atual dos TADs.
- LIMPA_HOST <host> tem tempo de leitura **O(h)**, com **h** sendo o tamanho do argumento host, pois representa a quantidade de caracteres na leitura dos parâmetros do comando. O tempo de execução será **O(t)**, em que **t** é o tamanho do host, ou seja, a quantidade de URLs que ele possui. É interessante observar que independe do tamanho das URLs do host, pois esse comando simplesmente libera ponteiros utilizados para cada URL. A complexidade de espaço de leitura e execução será **O(1)**, pois o espaço extra necessário para ambas independe dos parâmetros de entrada e do estado atual dos TADs.
- LIMPA_TUDO tem tempo de leitura **O(1)** (pois não possui argumento) e tempo de execução **O(m)** se **m** é a quantidade de URLs salvas no momento, pois o espaço alocado para cada um deles deverá ser desalocado. A complexidade de espaço de leitura e execução será **O(1)**, pois o espaço extra necessário para ambas independe dos parâmetros de entrada e do estado atual dos TADs.

Estratégias de Robustez

A principal estratégia de robustez do programa é conferir se as entradas de cada função são válidas sempre que possível. Isso garante que qualquer possível erro na implementação seja detectado facilmente, pois é esperado que, se os erros não foram detectados no início da execução da função, então não haverão erros depois. Também é importante para evitar coisas como não desalocar pedaços da memória no *heap* que não serão mais usados pelo resto da execução do programa. Essa validação é feita nas principais funções do programa, utilizando as macros *avisoAssert* e *erroAssert*, sendo que a segunda interrompe a execução do programa além de avisar o erro, a primeira apenas imprime o aviso.

Utilizamos o *erroAssert* quando caso as condições especificadas não sejam atendidas, o comportamento do programa não é definido. Um exemplo é na função *escalona*, pois como não é definido o escalonamento de uma quantidade negativa de URLs, não faz sentido o programa continuar a execução. Utilizamos essa macro nos seguintes casos:

- Não foi possível ler o arquivo de entrada ou escrever no arquivo de saída.
- Não for possível escrever no arquivo de registro de desempenho;
- O método *get_host* é chamada para uma instância de *Comando* que não possui host como argumento;
- O método *more_urls* é chamada para uma instância de *Comando* que não possui ulrs como argumento;
- O método *get_url* é chamada para uma instância de *Comando* que não possui ulrs como argumento;
- O método *get_quantidade* é chamada para uma instância de *Comando* que não possui quantidade como argumento;
- O operador de leitura de stream *operator>>* de *Comando* lê uma quantidade negativa em algum argumento;
- O método *escalona_host* é chamado para para uma quantidade negativa como argumento;
- O método *escalona_host_interno* é chamado para para uma quantidade negativa como argumento, ou para um nó de host nulo;

- O método *limpa_host* é chamado para um host que não está salvo (e portanto não pode ser limpo).
- A função *get_nome_saida* é chamada para um nome de entrada que não tem extensão (e assim não é possível gerar o nome do arquivo de saída).
- O método *get_host_string* é chamado para uma instância de URL que é pequena demais para ter “http://” (o que já é um estado inválido por si só).
- O método *extensao_valida* recebe um URL de tamanho inválido (menor que 5, insuficiente para ter o “http://”, que configura um erro por si só), ou um URL que possui fragmento.
- O método *remove_www* recebe um url pequeno demais, de tamanho insuficiente para conter “http://”, o que por si só configura um url inválido.

Já o avisoAssert seria utilizado quando mesmo que as condições não sejam atendidas, o comportamento do programa continua definido, mas algo descessário poderia ser removido. Nesse programa, essaa macro não é utilizada.

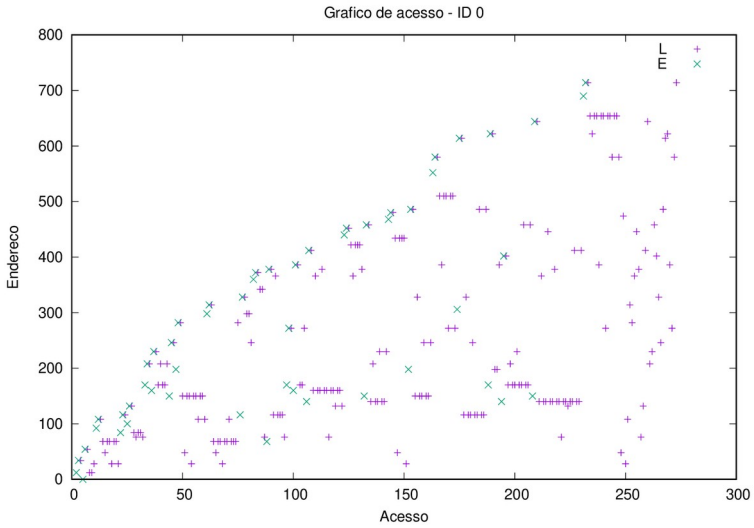
Análise Experimental

Para a análise experimental, utilizamos matrizes de entrada de teste de diferentes tamanhos, observando os resultados dos arquivos de registro considerando separadamente os padrões de acesso à memória e o tempo de execução. Essa separação é importante porque o tempo necessário para ler e escrever no arquivo de registro pode ser considerável, afinal estamos fazendo acessos à memória secundária.

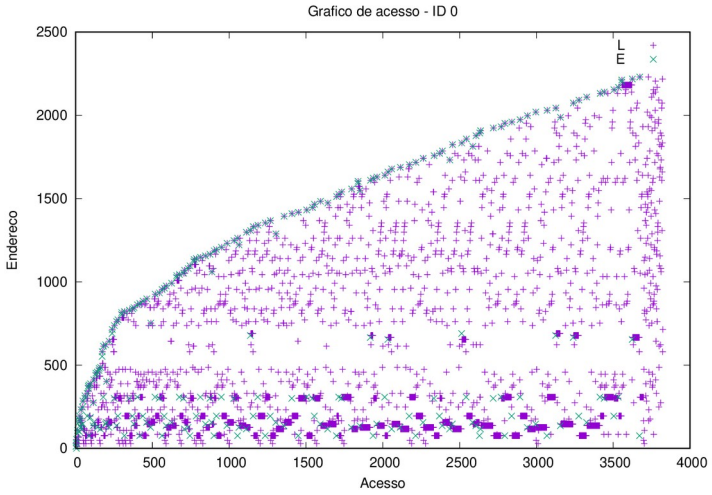
Para termos uma análise mais precisa, utilizamos o gerador de carga disponibilizado no Moodle da disciplina, o “geracarga”. Foram gerados pacotes de 100 hosts, cada um com 100 URLs, de variâncias e profundidades variadas, além de pacotes de de 50, 25 e 5 hosts e quantidade de URLs variando de 5 a 100 na mesma ordem. Apresentamos mapas de acesso à memória pra cada um desses casos, além dos tempos tomados pelo program a em cada uma das situações. O código de distância de pilha foi uma versão levemente modificada (trocando tkvet[5] por tkvet[4], pela forma como esse programa foi feito) da disponibilizada pelo professor.

Testes realizados com 5 servidores:

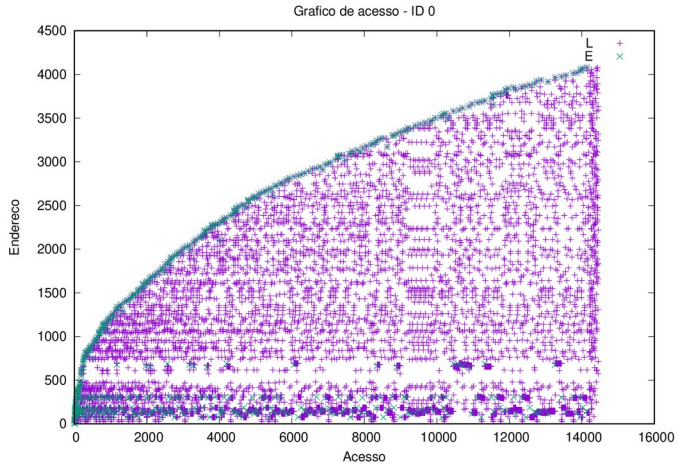
- 5 servidores e 5 URLs por host:
  - Tempo de duração: 0.0003585350059438497 segundos
  - Exemplo de mapa de acesso à memória:



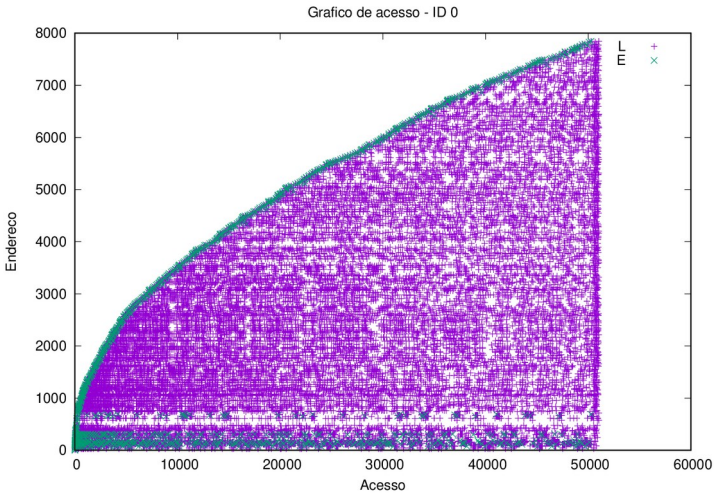
- 5 servidores e 25 URLs por host:
  - Tempo de duração: 0.0016802689933683723 segundos
  - Exemplo de mapa de acesso à memória:



- 5 servidores e 50 URLs por host:
  - Tempo de duração: 0.005648432001180481 segundos
  - Exemplo de mapa de acesso à memória:

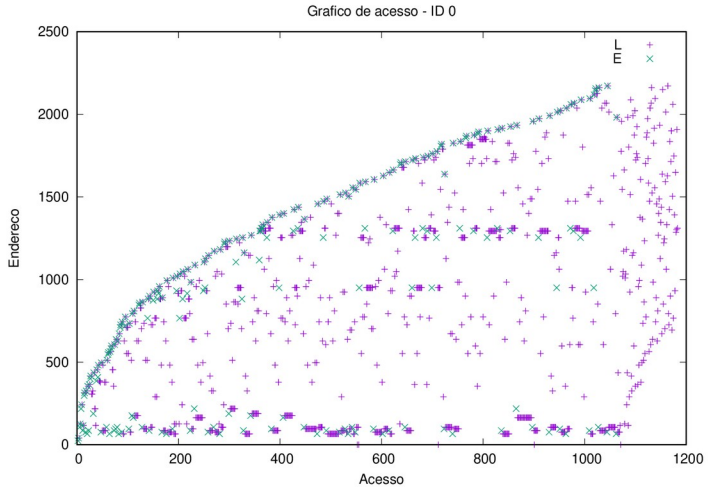


- 5 servidores e 100 URLs por host:
  - Tempo de duração: 0.019190635000995826 segundos
  - Exemplo de mapa de acesso à memória:

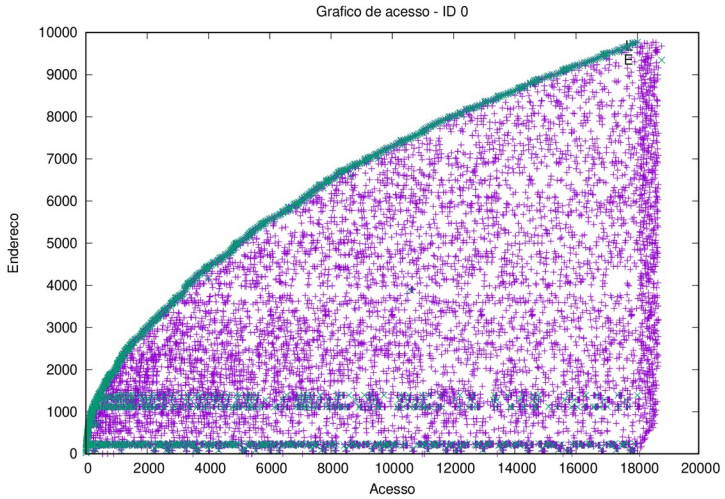


Testes realizados com 25 servidores:

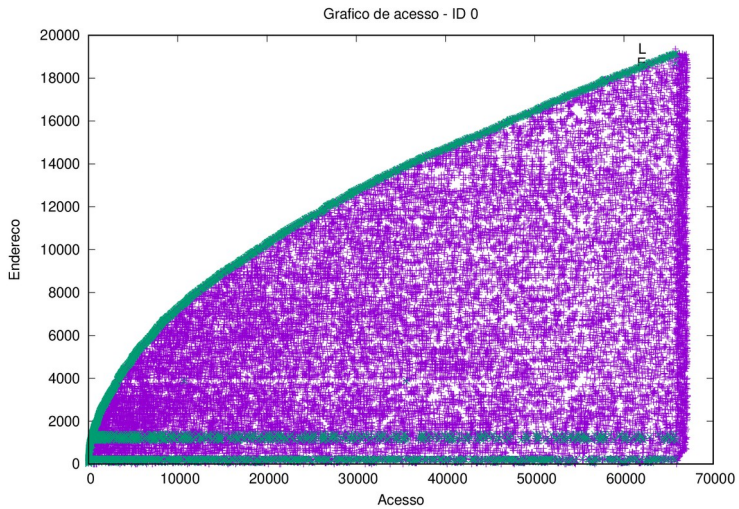
- 25 servidores e 5 URLs por host:
  - Tempo de duração: 0.011294769996311516 segundos
  - Exemplo de mapa de acesso à memória:



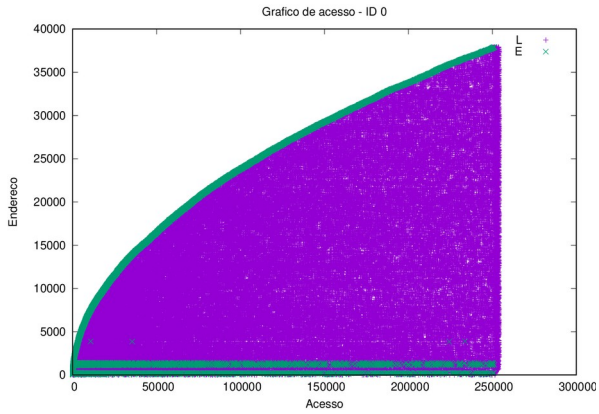
- 25 servidores e 25 URLs por host:
  - Tempo de duração: 0.0395611710000594 segundos
  - Exemplo de mapa de acesso à memória:



- 25 servidores e 50 URLs por host:
  - Tempo de duração: 0.052124137999953746 segundos
  - Exemplo de mapa de acesso à memória:

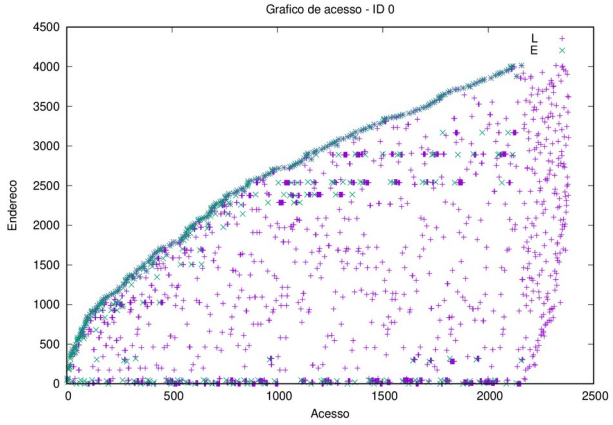


- 25 servidores e 100 URLs por host:
  - Tempo de duração: 0.1065367680057534 segundos
  - Exemplo de mapa de acesso à memória:

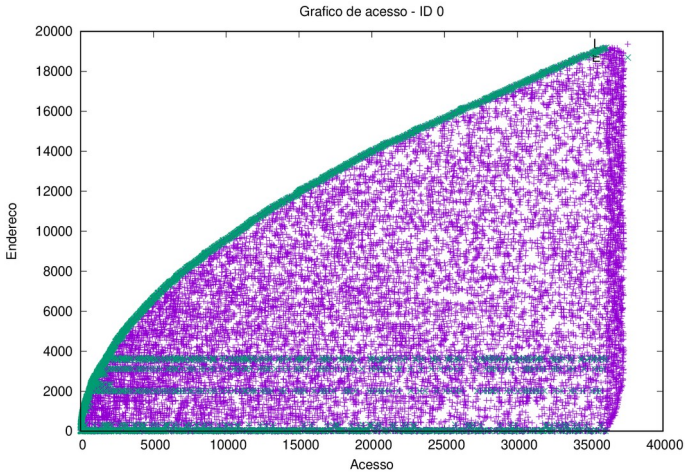


Testes realizados com 50 servidores:

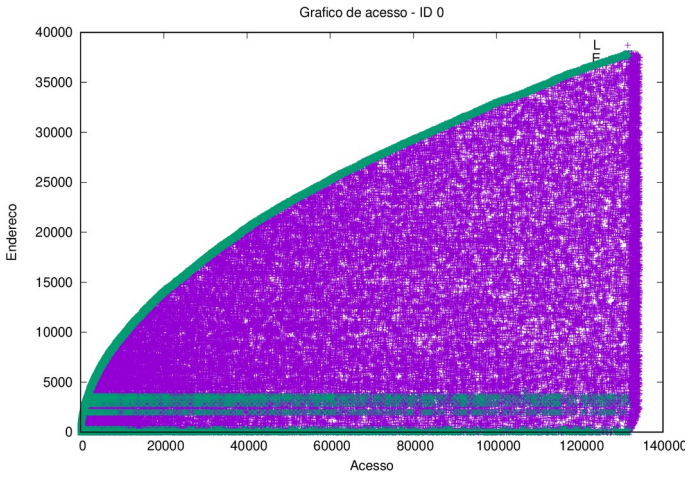
- 50 servidores e 5 URLs por host:
  - Tempo de duração: 0.029639438999993217 segundos
  - Exemplo de mapa de acesso à memória:



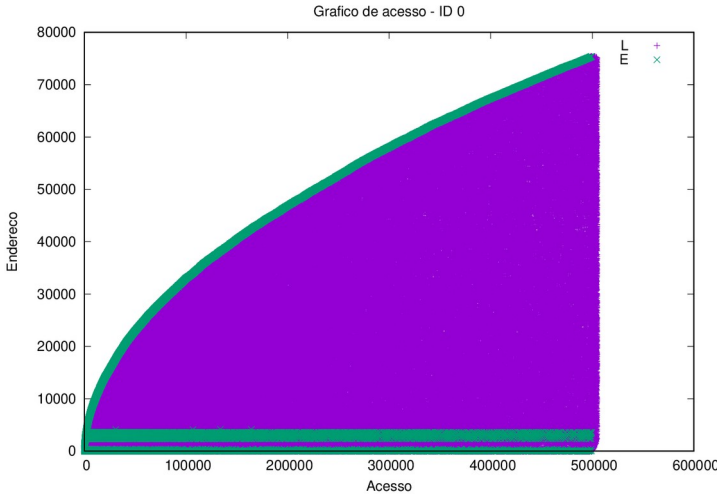
- 50 servidores e 25 URLs por host:
  - Tempo de duração: 0.05061228699992171 segundos
  - Exemplo de mapa de acesso à memória:



- 50 servidores e 50 URLs por host:
  - Tempo de duração: 0.08913778700002695 segundos
  - Exemplo de mapa de acesso à memória:

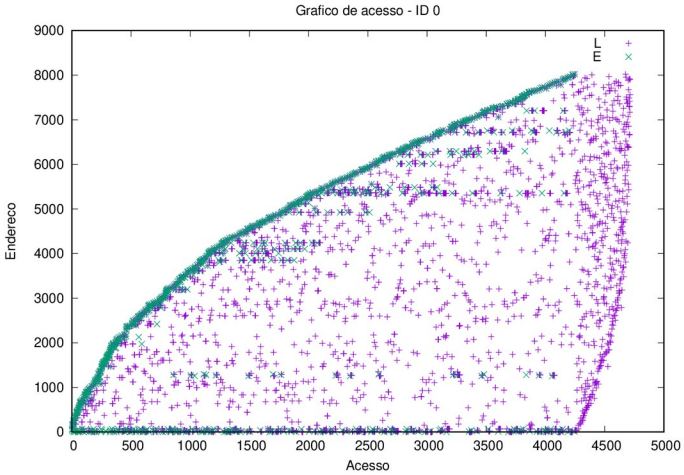


- 50 servidores e 100 URLs por host:
  - Tempo de duração: 0.21509063200005585 segundos
  - Exemplo de mapa de acesso à memória:

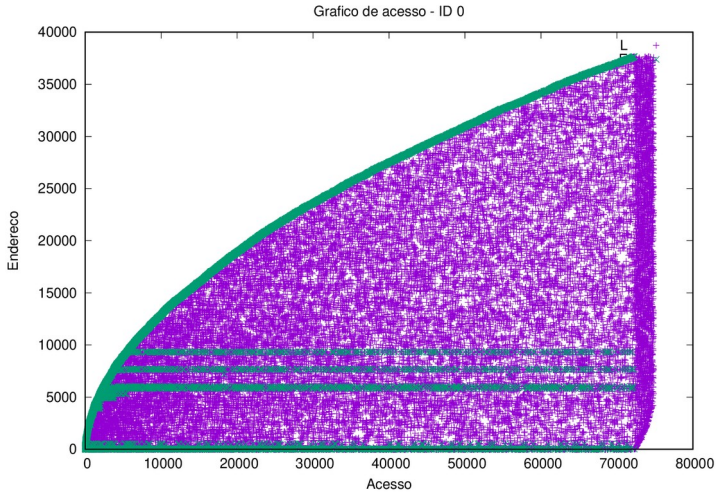


Testes realizados com 100 servidores:

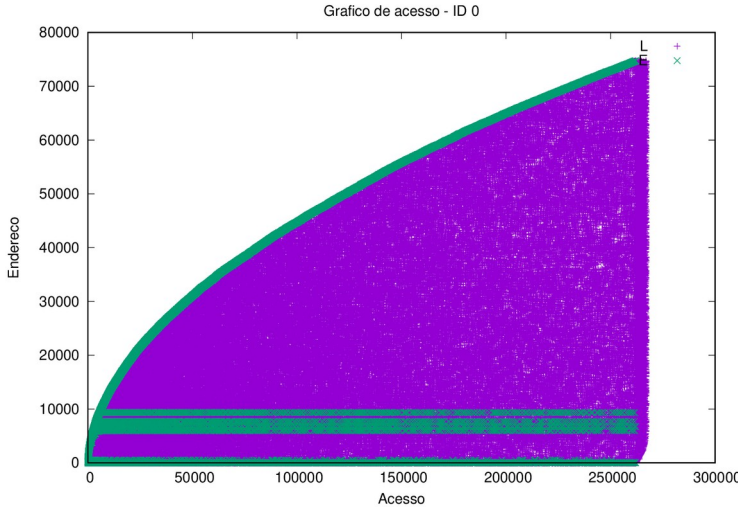
- 100 servidores e 5 URLs por host:
  - Tempo de duração: 0.039051302999951076 segundos
  - Exemplo de mapa de acesso à memória:



- 100 servidores e 25 URLs por host:
  - Tempo de duração: 0.07238368499997705 segundos
  - Exemplo de mapa de acesso à memória:



- 100 servidores e 50 URLs por host:
  - Tempo de duração: 0.1434476789997916 segundos
  - Exemplo de mapa de acesso à memória:



- 100 servidores e 100 URLs por host:
  - Tempo de duração: 0.4622866669999439 segundos
  - Exemplo de mapa de acesso à memória:

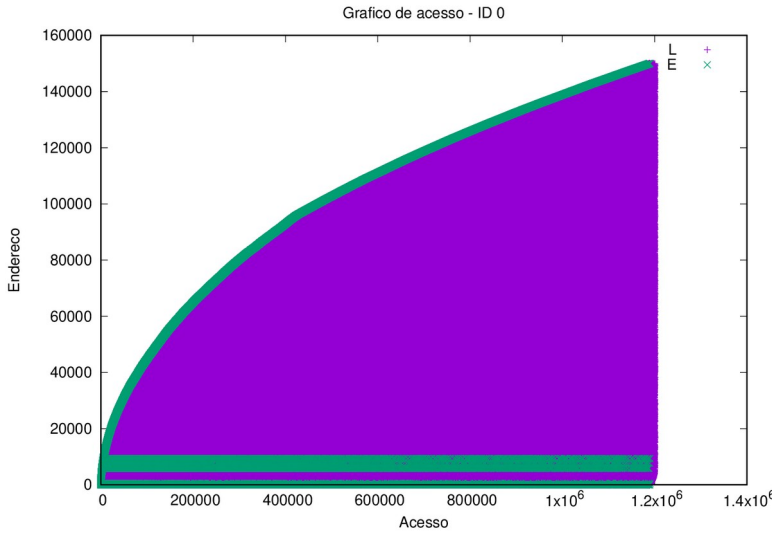


Gráfico de tempo de execução em função da quantidade total de URLs:

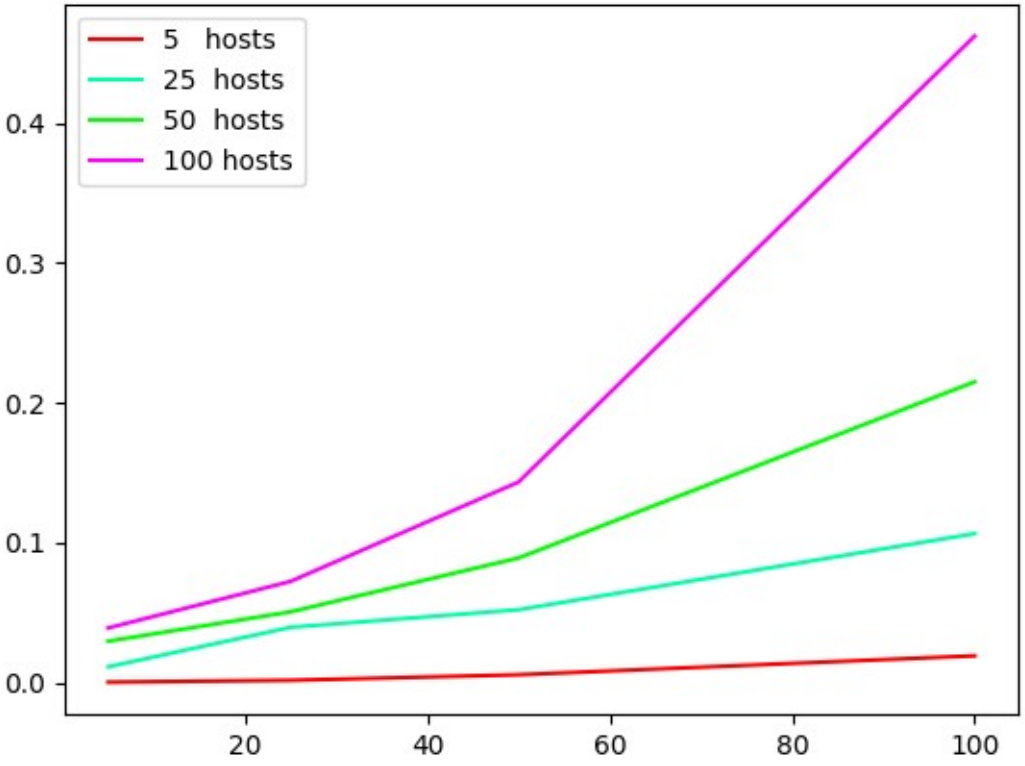
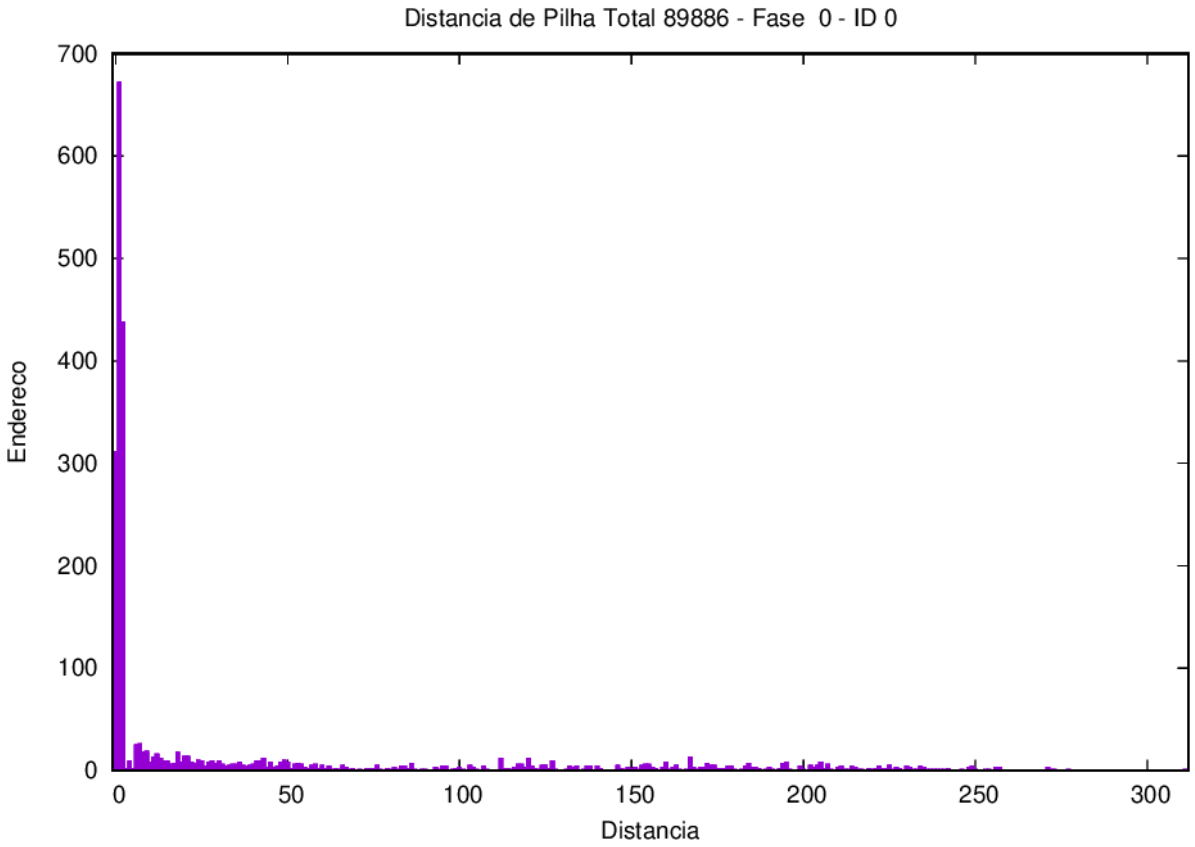


Gráfico de distância de pilha pra 50 hosts e 5 URLs:



Conclusões

Podemos concluir, portanto, que existe um padrão de acesso à memória no programa em que locais próximos a um recentemente acessado tem maior probabilidade de serem acessados em um futuro próximo, apesar de todas as alocações serem feitas dinamicamente. Como esperado, conforme a quantidade de URLs cresceu, o tempo de execução cresceu significativamente mais do que o crescimento da quantidade de entradas das mesmas (veja a seção *Análise de Complexidade* para ver o porque).

Podemos, portanto, afirmar que a análise prévia dos algoritmos e estruturas de dados que serão utilizadas contribui de forma muito eficiente para prever como o programa se comportará quando estiver pronto, sendo uma etapa fundamental no processo de desenvolvimento.

Bônus: Os Desafios

Esse trabalho implementa também duas versões alternativas, com executáveis “desafio1” e “desafio2”:

- *desafio1* passa pelos hosts utilizando a estratégia de Breadth-First-Search: ao invés de escalonarmos todos os hosts de um URL antes de ir para o próximo, escalonamos um de cada URL, priorizando assim a diversidade de hosts. Utilizamos o arquivo de implementação escalonador_desafio1 para esse desafio e mesmos arquivos de cabeçalho e outros de implementação, visando assim usufruir da flexibilidade permitida pelo C++.
- *desafio2* passa pelos hosts utilizando a estratégia de Best-First-Search: olhamos primeiro hosts com mais URLs, o que muda dinamicamente enquanto escalonamos. *Note que o comando de listar hosts ainda lista na ordem original*, como pedido na espeficicação do trabalho. Para implementar o desafio, criamos os arquivos *fila_hosts_desafio2.cpp*, *escalonador_desafio2.cpp* e *fila_hosts_desafio2.cpp*.

Bibliografia

CPP Reference. Disponível em: https://www.cplusplus.com/reference/. Acesso em: 01 nov. 2021.
C Programming Language DevDocs. Disponível em: https://devdocs.io/c/. Acesso em: 01 nov. 2021.
STACKOVERFLOW. Disponível em: https://stackoverflow.com/. Acesso em: 01 nov. 2021.
CORMEN, Thomas H. *et al*. **Introduction To Algorithms**. 3. ed. Cambridge: The Mit Press, 2009.

Instruções para compilação e execução

Compilação:

Para compilar o programa, é preciso chamar o programa *make* na pasta TP, que utilizará o M*akefile* lá presente para compilar todas as partes do programa que foram atualizadas desde a última compilação, e uni-las. Apesar de não ser tão relevante nesse caso, ainda é possível observar a utilidade dessa separação quando reutilizamos a maioria das classes e suas implementações nos desafios, e mais ainda quando temos projetos grandes: nesse caso é muito importante ter o *Makefile* para que não seja preciso recompilar todo o projeto para cada pequena mudança que fizermos.

Execução:

Para executar o programa, é preciso chamar o executável armazenado na pasta *bin* com nome *escalonar*. É preciso passar para o programa o seguinte argumento:

- *<arq>* para informar o caminho do arquivo de entrada do programa. Note que esse nome será adaptado para gerar o nome de saída e o nome de registro do padrão de acesso à memória.

  Além de ser possível informar o seguinte argumento opcional:
- *-l* para informar se é desejado registrar os acessos à memória. Caso essa opção não seja passada, será registrado somente o tempo de execução.

É importante destacar também que o arquivo de entrada do programa devem seguir a seguinte especificação:

- O nome do arquivo deve possuir alguma extensão (terminar em .txt, .texto, .etc …).
- Cada linha do arquivo deve possuir exatamente um comando e seus argumentos, ou um *URL*.
- Caso uma linha possua um comando diferente de ADD_URLS, ou esta é a última linha do arquivo, ou a linha seguinte é um comando.
- Caso uma linha possua um comando ADD_URLS, as próximas *<quantidade>* linhas informadas como argumento do comando devem possuir um *URL* cada.
- Caso uma linha possua um *URL*, esse URL deve ser parte de um bloco de URLs informados por um comando ADD_URLS.
- Caracteres de espaço e tab extras serão ignorados.

Executáveis gerados:

- *escalonar* implementa a versão do escalonador com estratégia Depth-First-Search, que escalona todos URLs de um host antes de ir para o próximo.
- *desafio1* implementa a versão do escalonador com estratégia Bredth-First-Search, que escalona a maior quantidade de hosts possível a cada etapa, valorizando a variedade.
- *desafio2* implementa a versão do escalonador com estratégia Best-First-Search, que escalona os hosts com mais URLs primeiro.