Trabalho Prático 2 de Estrutura de Dados

Artur Gaspar da Silva, 2020006388

Introdução

Este arquivo visa documentar e analisar o funcionamento do programa “ordena”, como terceiro 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 são lidos em um arquivo especificado como argumento da linha de comando e salvando os resultados em outro arquivo, também especificado como argumento na linha de comando. No entanto, a saída é ordenada em pacotes (que chamamos de rodadas).

Na versão principal, os URLs são ordenados a partir do algoritmo quicksort (implementado de forma recursiva) e armazenados em arquivos de rodada, que posteriormente são intercalados, sendo priorizados URLs com mais acessos. Incluímos também uma análise da complexidade do programa em diversos casos, e uma análise experimental do desempenho do mesmo. Os desafios (que implementam outras maneiras de intercalar as rodadas) também podem ser acessados a partir dos argumentos da linha de comando.

Método

Implementamos doze tipos de dados distintos: “Fila_Par_Int”, “Heap_URL_Acessos”, “Heapsort_Recursivo”, “Mergesort_Nao_Recursivo”, “Mergesort_Recursivo”, “Ordenador”, “Pilha_Par_Int”, “Quicksort_Nao_Recursivo”, “Quicksort_Recursivo”, “Rodada_Intercalator”, “Rodada_Manipulator”, “URL_Acessos”. Temos também métodos internos de cada tipo de dados, seis macros, uma função para identificar os argumentos da linha de comando, 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 todos os URLs a ordenar e seus respectivos números de acessos, e o arquivo de saída contém os URLs ordenados com seus números de acessos. Ao longo dessa documentação, chamamos de **elemento** uma dupla URL e seu número de acessos, ou de um *URL_Acessos*.
- Ambos arquivos possuem exatamente um URL por linha, seguido de um espaço e um número inteiro representando a quantidade de acessos.
- Logo após o número de acessos de cada URL, deve haver um caractere de quebra de linha que representa o fim da linha atual.

Algoritmo de ordenação (desafios):

- Caso não haja opção “-d”, o programa ordena os URLs nas rodadas utilizando o algoritmo de quicksort implementado de forma recursiva.
- Caso seja passada a opção “-d 1_merge”, o programa utiliza o algoritmo de mergesort implementado de forma recursiva (desafio 1).
- Caso seja passada a opção “-d 1_heap”, o programa utiliza o algoritmo de heapsort (desafio 1).
- Caso seja passada a opção “-d 2_quick”, o programa utiliza o algoritmo de quicksort implementado de forma iterativa (desafio 2).
- Caso seja passada a opção “-d 2_merge”, o programa utiliza o algoritmo de mergesort implementado de forma iterativa (desafio 2).

Atributos e métodos do tipo “Fila_Par_Int”:

O tipo “Fila_Par_Int” é especificado no arquivo de cabeçalho “fila_par_int.hpp”, possuindo seis atributos privados:

- *capacidade* representa a capacidade máxima da fila (definida durante a criação da fila, não muda durante a execução do programa).
- *quantidade* representa a quantidade de elementos atualmente presente na fila.
- *primeiro* representa o arranjo que guarda os primeiros inteiros de cada par.
- *segundo* representa o arranjo que guarda os segundos inteiros de cada par.
- *pos_init* representa a posição inicial da fila no arranjo real (implementada como arranjo circular, portanto é um atributo necessário).
- *pos_final* representa a posição final da fila no arranjo real (implementada como arranjo circular, portanto é um atributo necessário).

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

- *Fila_Par_Int* constrói uma fila a partir da sua capacidade máxima.
- *adiciona* adiciona um par de inteiros à fila.
- *primeiro_a* retorna o primeiro inteiro do primeiro par da fila.
- *primeiro_b* retorna o segundo inteiro do primeiro par da fila.
- *tira_primeiro* remove o primeiro par da fila.
- *vazia* informa se a fila está vazia.
- *destruir* desaloca toda a memória alocada para a fila.
- *~Fila_Par_Int* efetivamente chama *destruir*.

Atributos e métodos do tipo “Heap_URL_Acessos”:

O tipo “Heap_URL_Acessos” é especificado no arquivo de cabeçalho “heap_url_acessos.hpp”, possuindo quatro atributos privados:

- *capacidade* informa a capacidade máxima do heap.
- *quantidade* informa a quantidade de elementos no heap.
- *heap* representa o arranjo que implementa o heap.
- *rodadas* representa um arranjo paralelo ao *heap*, que armazena a rodada de cada entrada (necessário para o algoritmo de intercalar).

No arquivo “heap_url_acessos .cpp” implementamos os métodos públicos e o método privado “refaz” usados para manipular o tipo “Heap_URL_Acessos”:

- *Heap_URL_Acessos* constrói um heap a partir de sua capacidade máxima.
- *adicionar* adiciona um elemento ao heap (consistindo de um URL_Acessos e sua rodada).
- *url_acessos_primeiro* retorna o maior elemento do heap (maior *URL_Acessos*).
- *rodada_primeiro* retorna a rodada de onde veio o maior elemento do heap (maior *URL_Aceesos*).
- *tira_primeiro* remove o maior elemento do heap e o reajusta para que ainda seja um heap.
- *vazio* informa se o heap está vazio.
- *~Heap_URL_Acessos* destrói o heap, liberando toda a memória alocada.
- *refaz* é uma função auxiliar na manutenção das propriedades de heap da estrutura. Efetivamente refaz um “galho” da árvore, mantendo o primeiro elemento o maior.

Atributos e métodos do tipo “Heapsort_Recursivo”:

O tipo “Heapsort_Recursivo” é especificado no arquivo de cabeçalho “heapsort_recursivo.hpp”, herdando do tipo “Ordenador”, e com os seguintes métodos:

- *Heapsort_Recursivo* é o construtor padrão (vazio).
- *ordena* é responsável por ordenar os *URL_Acessos* de um arranjo qualquer (sobrescrito de *Ordenador*) a partir do algoritmo de heapsort.

Atributos e métodos do tipo “Mergesort_Nao_Recursivo”:

O tipo “Mergesort_Nao_Recursivo” é especificado no arquivo de cabeçalho “mergesort_nao_recursivo.hpp”, herdando do tipo “Ordenador”, e com os seguintes métodos:

- *Mergesort_Nao_Recursivo* é o construtor padrão (vazio).
- *ordena* é responsável por ordenar os *URL_Acessos* de um arranjo qualquer (sobrescrito de *Ordenador*) a partir do algoritmo mergesort implementado de forma não recursiva.

Atributos e métodos do tipo “Mergesort_Recursivo”:

O tipo “Mergesort_Recursivo” é especificado no arquivo de cabeçalho “mergesort_recursivo.hpp”, herdando do tipo “Ordenador”, e com os seguintes métodos:

- *Mergesort_Recursivo* é o construtor padrão (vazio).
- *ordena* é responsável por ordenar os *URL_Acessos* de um arranjo qualquer (sobrescrito de *Ordenador*) a partir do algoritmo de mergesort implementado de forma recursiva.

Atributos e métodos do tipo “Ordenador”:

O tipo “Ordenador” é especificado no arquivo de cabeçalho “ordenador.hpp”, possuindo dois atributos privados:

- *lista_urls* representa o arranjo de *URL_Acessos* a ser ordenado.
- *numero_urls* representa a quantidade de elementos no arranjo a ser ordenado.

No arquivo “ordenador.cpp” implementamos os métodos públicos usados para manipular o tipo “Ordenador” (exceto o método ‘ordena’, que é sobrescrito pelas subclasses do Ordenador):

- *Ordenador* é o construtor padrão (vazio) do tipo.
- *set_fonte* registra o ponteiro pro arranjo de *URL_Acessos* que será ordenado.
- *ordena* é um método virtual vazio que será sobrescrito pelos tipos que herdam do *Ordenador*.
- *~Ordenador* é o destrutor padrão do tipo. Note que ele não desaloca o espaço alocado para os arranjos porque ele não é o tipo que aloca esse espaço em primeiro lugar, apenas ordenando os elementos de lá.

Atributos e métodos do tipo “Pilha_Par_Int”:

O tipo “Pilha_Par_Int” é especificado no arquivo de cabeçalho “pilha_par_int.hpp”, possuindo quatro atributos privados:

- *capacidade* informa a capacidade atual da pilha. Pode mudar durante a execução do programa.
- *quantidade* informa a quantidade atual de elementos na pilha.
- *primeiro* representa o arranjo em que serão armazenados os primeiros elementos de cada par da pilha.
- *segundo* representa o arranjo em que serão armazenados os segundos elementos de cada par da pilha.

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

- *Pilha_Par_Int* é o construtor padrão do tipo, que recebe uma capacidade inicial de espaço alocado (pode aumentar).
- *adiciona* adiciona um par de elementos ao topo da pilha.
- *topo_a* retorna o primeiro elemento do primeiro par do tipo da pilha.
- *topo_b* retorna o segundo elemento do primeiro par do topo da pilha.
- *tira_topo* remove o elemento do topo da pilha.
- *vazia* informa se a pilha está vazia.

- *destruir* limpa todo o espaço alocado para a pilha.
- *~Pilha_Par_Int* é o destrutor padrão da pilha, e efetivamente chama *destruir*.

Atributos e métodos do tipo "Quicksort_Nao_Recursivo":

O tipo "Quicksort_Nao_Recursivo" é especificado no arquivo de cabeçalho "quicksort_nao_recursivo.hpp", herdando do tipo "Ordenador", e com os seguintes métodos:

- *Quicksort_Nao_Recursivo* é o construtor padrão (vazio).
- *ordena* é responsável por ordenar os *URL_Acessos* de um arranjo qualquer (sobrescrito de *Ordenador*) a partir do algoritmo de quicksort implementado de forma não recursiva.

Atributos e métodos do tipo "Quicksort_Recursivo":

O tipo "Quicksort_Recursivo" é especificado no arquivo de cabeçalho "quicksort_recursivo.hpp", herdando do tipo "Ordenador", e com os seguintes métodos:

- *Quicksort_Recursivo* é o construtor padrão (vazio).
- *ordena* é responsável por ordenar os *URL_Acessos* de um arranjo qualquer (sobrescrito de *Ordenador*) a partir do algoritmo de quicksort implementado de forma recursiva.

Atributos e métodos do tipo "Rodada_Intercalator":

O tipo "Rodada_Intercalator" é especificado no arquivo de cabeçalho "rodada_intercalator.hpp", possuindo quatro atributos privados:

- *fitas* representa os arquivos das rodadas de onde serão lidos os *URL_Acessos* a intercalar.
- *nome_saida* registra o nome do arquivo de saída final.
- *heap_url_acessos* representa o heap de *URL_Acessos* que será utilizado no processo de intercalar os elementos dos arquivos de rodada pro arquivo final de saída.
- *num_rodadas* informa o número de rodadas utilizadas no total no processo de intercalar.

No arquivo "rodada_intercalator.cpp" implementamos os métodos públicos usados para manipular o tipo "Rodada_Intercalator":

- *Rodada_Intercalator* é o construtor padrão do tipo, que recebe o número de rodadas existente e o nome do arquivo final de saída.
- *intercalar* efetivamente intercala os elementos dos arquivos de rodada e armazena o resultado no arquivo final de saída.
- *destruir* limpa a memória alocada dinamicamente pelo objeto.
- *~Rodada_Intercalator* é o destrutor padrão do tipo, que efetivamente chama *destruir*.

Atributos e métodos do tipo "Rodada_Manipulator":

O tipo "Rodada_Manipulator" é especificado no arquivo de cabeçalho "rodada_manipulator.hpp", possuindo quatro atributos privados:

- *ord* é o *Ordenador* utilizado, efetivamente definindo qual será o algoritmo de ordenação de cada rodada.
- *arq_entrada* é o arquivo de entrada de onde serão lidos os elementos.
- *num_rodadas* é o número de máximo rodadas que serão utilizadas.
- *n_mem_prim* é o número de elementos que cabem na memória primária (efetivamente a quantidade de URLs a ordenar por rodada).

No arquivo "rodada_manipulator.cpp" implementamos os métodos públicos usados para manipular o tipo "Rodada_Manipulator":

- *Rodada_Manipulator* é o construtor padrão do tipo, recebendo o nome de entrada, o *Ordenador*, o número de rodadas e o número de elementos por rodada.
- *gera_rodadas* gera os arquivos de rodada até o limite autorizado.
- *intercala_rodadas* intercala os arquivos de rodada, armazenando os URLs e seus acessos de forma completamente ordenada no arquivo de saída.

- *acabou* informa se ainda há elementos a ler do arquivo de entrada.
- *destruir* limpa a memória alocada pelo manipulador.
- *~Rodada_Manipulator* é o destrutor padrão do tipo, que efetivamente chama *destruir*.

Atributos e métodos do tipo "URL_Acessos":

O tipo "URL_Acessos" é especificado no arquivo de cabeçalho "url_acessos.hpp", possuindo dois atributos públicos:

- *url* é o URL do tipo, armazenado na forma de string.
- *quantidade* é a quantidade de acessos do URL, armazenado na forma de string.

No arquivo "url_acessos .cpp" implementamos o método público usado para construir instâncias do tipo "URL_Acessos":

- *URL_Acessos* é o construtor padrão do tipo (vazio).

Por fim, no mesmo arquivo, implementamos também os operadores ">>", "<<", e "<":

- *operator<* compara dois *URL_Acessos*, efetivamente comparando as quantidades de acessos. Caso dois *URL_Acessos* tenham o mesmo número de acessos, comparamos os URLs lexicograficamente, ou seja, utilizando ordem alfabética.
- *operator>>* lê um elemento do tipo de uma *istream*, simplesmente lendo uma string e um inteiro, armazenando-os em *url* e *quantidade*, respectivamente.
- *Operator<<* escreve um elemento do tipo em uma *ostream*, simplesmente escrevendo *url* seguido de *quantidade*.

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 “ordena.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.
- *parse_args* é uma função que recebe os argumentos da linha de comando e os armazena em variáveis globais de forma a ficarem simples de acessar.
- *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 ordenação do .

Escolha de pivôs

Tanto o quicksort recursivo quanto o não-recursivo exigem a escolha de um pivô a cada etapa. A estratégia utilizada para essa escolha foi a da mediana de três: selecionamos um valor do começo do array, um do final, e um do meio. Em seguida, pegamos a posição do que possui valor igual à mediana dos três como pivô. Isso evita o pior caso original de sempre selecionar o maior ou menor valor, e diminui consideravelmente a probabilidade de cairmos num pior caso.

Análise de Complexidade

As principais tarefas realizadas pelo programa são: leitura de URLs e suas respectivas quantidades de acessos em pacotes, ordenamento desses pacotes, e intercalamento desses pacotes para o arquivo de saída (que possui todos URLs ordenados pelu número de acessos, e em caso de empate, por ordem alfabética). Abaixo faremos a análise de complexidade assintótica do programa nos principais casos possíveis na ordem em que são executados:

- Primeiramente os argumentos são processados por *parse_args*, e esse processamento passa por cada um dos argumentos da linha de comando, mesmo que hajam argumentos redundantes. Portanto, se **a** é a quantidade de argumentos, podemos dizer que a complexidade assintótica de tempo desse procedimento no pior caso é de **Θ(a)**, e no melhor caso é **Θ(1)** em relação ao número de argumentos: caso o primeiro argumento seja *-h*, o programa imprime a ajuda sobre quais argumentos são aceitos, e sai sem processar o resto dos argumentos. Caso todos argumentos sejam válidos e nenhum deles seja *-h*, temos o pior caso, em que o programa passa por cada um dos argumentos, mesmo que sejam redundantes. A complexidade de espaço dessa função é de **Θ(1)**, pois o espaço necessário em memória para sua execução independe da quantidade de argumentos passados.

- Após o processamento dos argumentos, 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, otempo 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 da quantidade de elementos (lembrando que chamamos um URL seguido de sua quantidade de acessos de um **elemento**, um *URL_Acessos*), e do algoritmo selecionado para ordenação. O registro de padrão de acesso e localidade influencia no tempo de execução mas essa influência não se observa assintoticamente.

Seja **r** a quantidade de rodadas total, **e** a quantidade máxima de elementos por rodada, e **n** a quantidade total de URLs:

- A primeira etapa do programa envolve ordenar **r** pacotes de **e** elementos com o algoritmo selecionado. Assintoticamente, todos os algoritmos utilizados (quicksort recursivo, quicksort não-recursivo, mergesort recursivo, mergesort não-recursivo, heapsort) possuem (para cada rodada) complexidade de tempo do caso médio de ordem **O(e*log(e))**, pois efetivamente ordenamos **e** elementos (fato visto em aula). O quicksort possui complexidade de pior caso **$O(e^2)$**, os outros possuem pior caso igual ao caso médio, e todos possuem melhor caso igual ao médio. Esses fatos foram todos vistos em aula. Todos os métodos recursivos (incluindo o heapsort) fazem uma quantidade logarítmica de chamadas recursivas e em cada uma alocam uma quantidade constante de memória. Já os métodos não-recursivos (mergesort e quicksort) pois deixam reservado espaço adicional linear na implementação atual (o quicksort por motivos de segurança, o mergesort porque a implementação utilizada primeiro separa todos os intervalos depois os processa, efetivamente fazendo o *merge* separadamente). Note que o método padrão é o quicksort recursivo, e os outros são dos desafios. No entanto, considerando o programa globalmente, em todos casos precisamos armazenar todos elementos em um arranjo antes de ordená-los, o que faz com que todos casos tenham complexidade de espaço **O(e)**. Em resumo, a primeira etapa (gerar rodadas) tem complexidade de tempo total **O(r*e*log(e))** e de espaço **O(e)**, tanto pro caso padrão quanto pra qualquer um dos desafios implementados.
- A segunda etapa do programa envolve intercalar os **r** pacotes para obter o arquivo final de **n** elementos ordenados. O método utilizado envolve armazenar o primeiro elemento de cada arquivo em um heap, removendo o primeiro elemento do heap e colocando outro do mesmo arquivo. Quando um arquivo acaba, nenhum elemento novo é colocado, e o processo está terminado quando o heap está vazio. Efetivamente, serão colocados e removidos **n** elementos no heap, e o heap nunca terá mais que **r** elementos, portanto a complexidade de tempo será **O(n log(r))**, pois cada adição e remoção consumirá **O(log(r))** operações. A complexidade de espaço é de **O(r)**, pois teremos no máximo **r** elementos no heap como já dito.

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 *tira_primeiro*, pois não é definido o que acontece quando se remove um elemetno de uma fila vazia, e não faz sentido o programa continuar a execução. Utilizamos essa macro nos seguintes casos:

- Argumento não passado na linha de comando ou passado de forma inválida (por exemplo, não foi informado o nome do arquivo de entrada, ou foi informada uma quantidade negativa de entidades por fita);
- Não foi possível ler o arquivo de entrada ou escrever no arquivo de saída;
- Não for possível escrever no arquivo de registro de desempenho;
- Não foi possível ler ou escrever em um dos arquivos de rodada;
- Um dos construtores *Fila_Par_Int*, *Heap_URL_Acessos* ou *Pilha_Par_Int* é chamado para capacidade máxima negativa;
- O método *set_fonte* é chamado para quantidade negativa de elementos;
- O construtor *Rodada_Intercalator* recebe número de rodadas não positivo;
- O construtor *Rodada_Manipulator* recebe número não positivo de rodadas ou de elementos aceitos na memória primária.
- O método *adiciona* é chamada para uma instância de *Fila_Par_Int*, *Heap_URL_Acessos* que já está cheia;
- Os métodos *primeiro_a*, *segundo_a*, ou *tira_primeiro*, é chamado para uma instância de *Fila_Par_Int*, *Heap_URL_Acessos*, *Pilha_Par_Int* que está vazia;
- O método *refaz* é chamado para posição não positiva ou para posição além do tamanho passado como parâmetro, tanto em *Heapsort_Recursivo* quanto em *Heap_URL_Acessos*;
- É encontrada uma tentativa de processar um intervalo inválido, em que o índice da esquerda não é menor que o da direita, tanto no *Mergesort_Nao_Recursivo*, quanto em *Mergesort_Recursivo*, *Quicksort_Nao_Recursivo*, *Quicksort_Recursivo*;
- Os limites passados como argumento na linha de comando de quantidade de arquivos de rodada e de quantidade de elementos na memória primária não permitem que o arquivo seja ordenado.

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, a macro é utilizada apenas para avisar que alguma opção está sendo ignorada:

- Caso a mesma opção seja passada duas vezes, apenas a última vez será considerada;
- Caso um desafio inválido seja passado como argumento, ele será ignorado.

Análise Experimental

Para a análise experimental, utilizamos entradas de teste de diferentes quantidades de URLs, observando os resultados dos arquivos de registro considerando separadamente as fases de geração das fitas e interpolação das mesmas (com diferença visível nos mapas de acesso, geraram gráficos de distância de pilha distintos), além da separação entre 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 10, 1000 e 100000 URLs, de variâncias e profundidades variadas, com 1, 5 e 20 rodadas (10 no caso de 10 URLs), e os algoritmos de quicksort, heapsort e mergesort, com quicksort e mergesort apresentando implementação não recursiva também. Apresentamos mapas de acesso à memória pra os casos de 10 e 1000 URLs, além dos tempos tomados pelo program em todas 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.

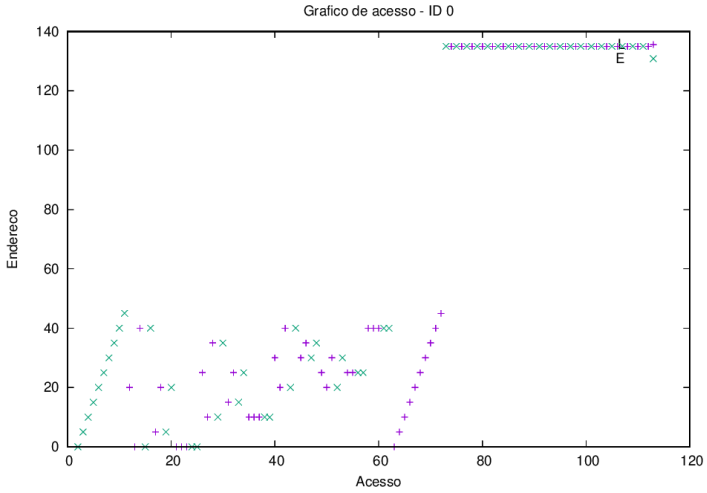
O tempo foi avaliado dez vezes para cada uma das entradas, e pegamos o tempo médio de execução desses dez. Temos mapas de acesso à memória que registram todos acessos diretos quaisquer de leitura e escrita, e mapas que registram apenas acessos realizados durante comparações. Os gráficos de distância de pilha são divididos por fase e se são de acesso ou comparação, mas a segunda fase é feita considerando tanto acessos gerais quanto de comparações. Para os desafios, apresentamos na documentação apenas um exemplo de mapa de acesso à memória e um histograma de distância de pilha, pois senão o PDF resultante excederia o tamanho máximo para entrega. No entanto, pode-se consultar todas as entradas, tempos, scripts e gráficos em https://drive.google.com/drive/folders/17MINcU0x1tP_h48zs-VK2Pyz0PFKyzWH?usp=sharing.

Especificações da máquina em que os experimentos foram realizados:

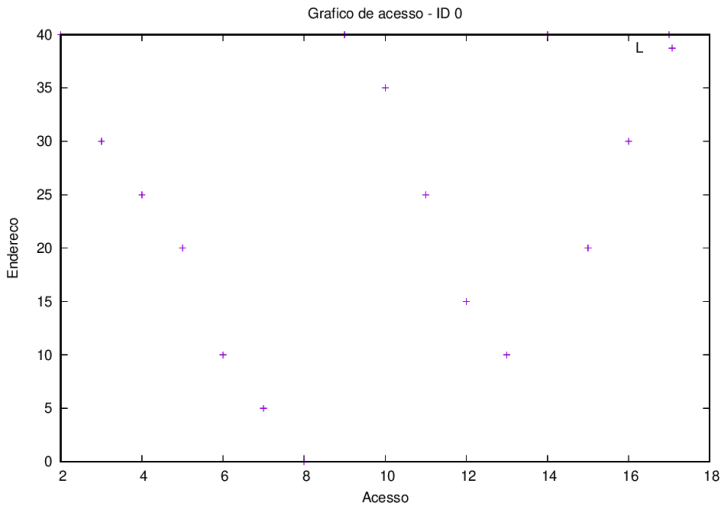
- Sistema Operacional Ubuntu 20.04.3 LTS
- Processador Intel Core I7-8565U CPU 1.80GHz x 8
- Placa de vídeo AMD Hainan / Mesa Intel UHD Graphics 620
- 7,6 Gb de memória (máquina com 8Gb de RAM)

Testes realizados com quicksort implementado recursivamente (TP2):

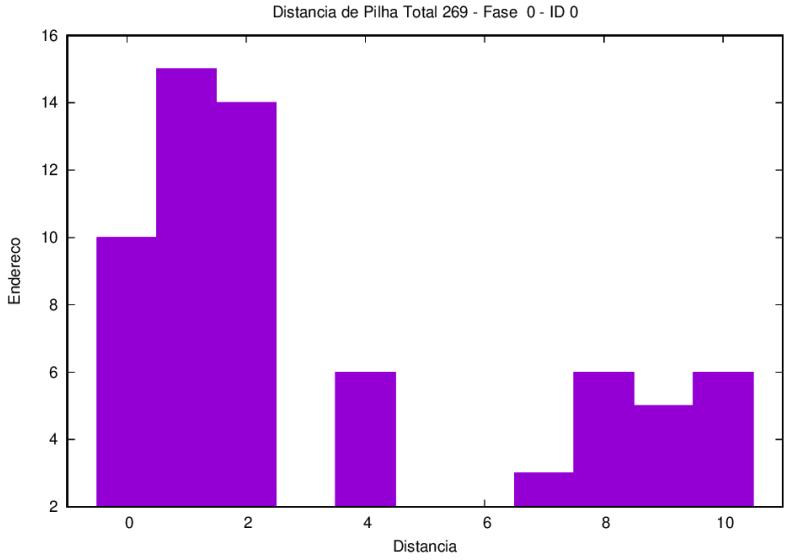
- 10 URLs e 1 rodada:
  - Tempo de duração médio: 0.0001676070999565127 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



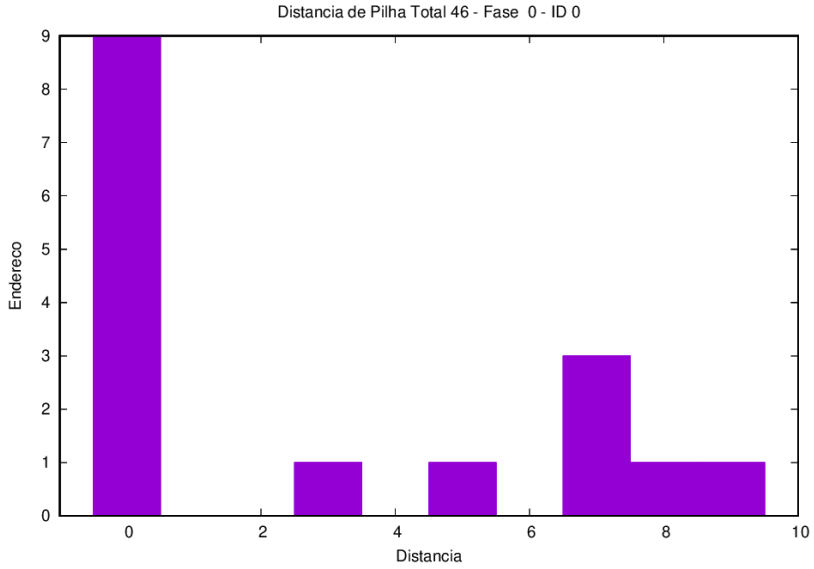
  - Exemplo de mapa de acesso à memória para comparações entre os elementos:



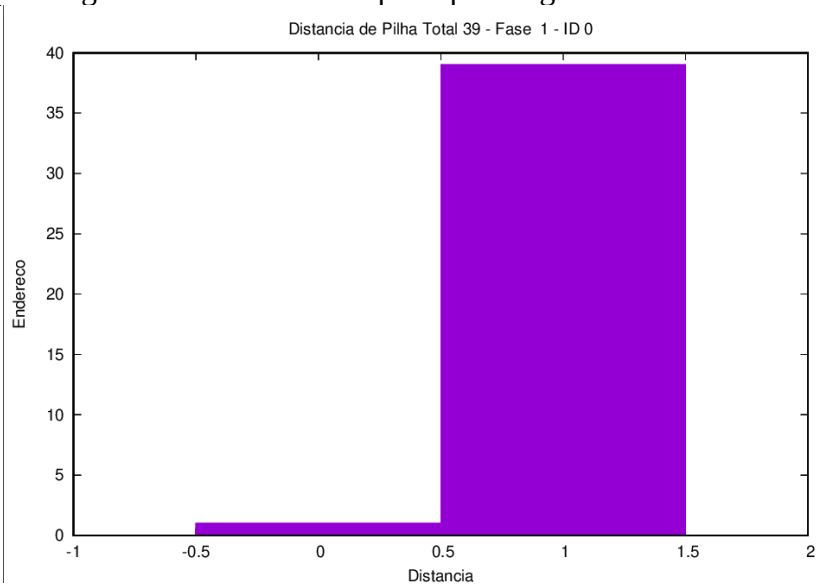
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



- Exemplo de gráfico de distância de pilha para primeira fase para comparações entre os elementos:

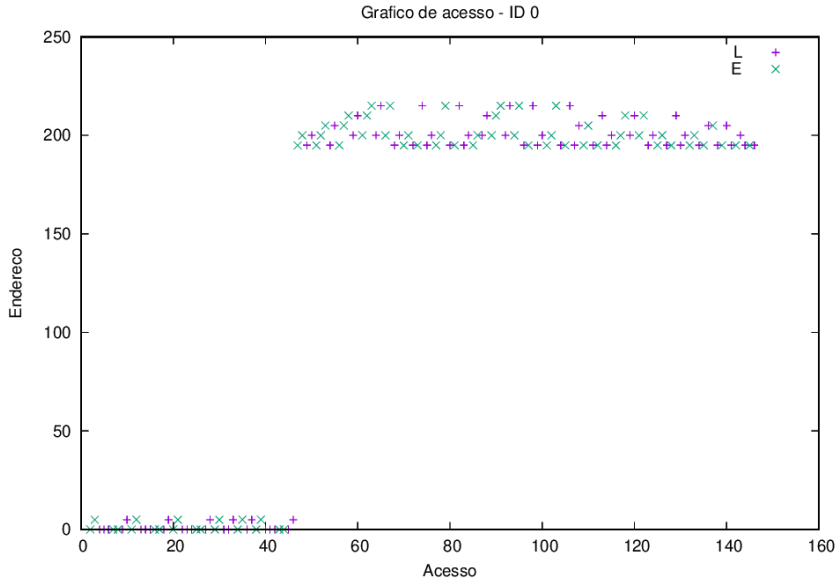


- Exemplo de gráfico de distância de pilha para segunda fase:

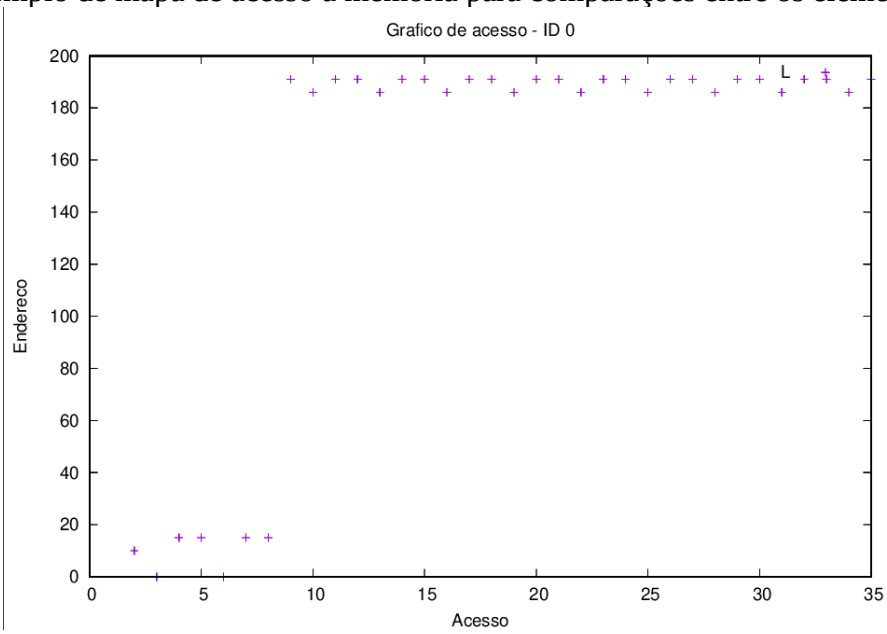


- 10 URLs e 5 rodadas:
  - Tempo de duração médio: 0.00027075840007455553 segundos

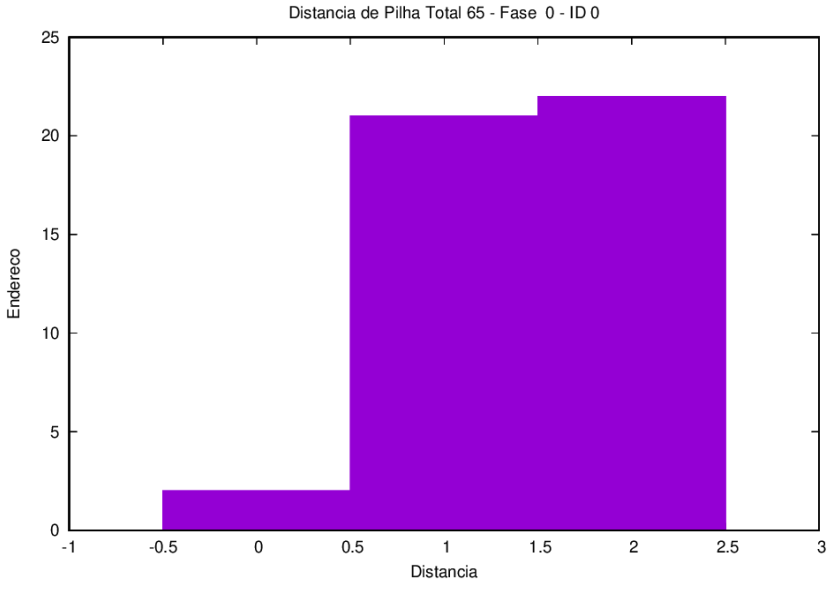
- Exemplo de mapa de acesso à memória para acessos aos elementos:



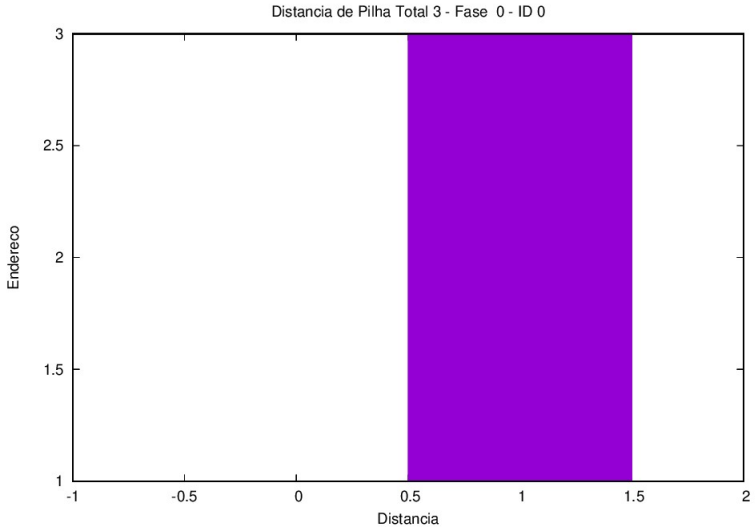
- Exemplo de mapa de acesso à memória para comparações entre os elementos:



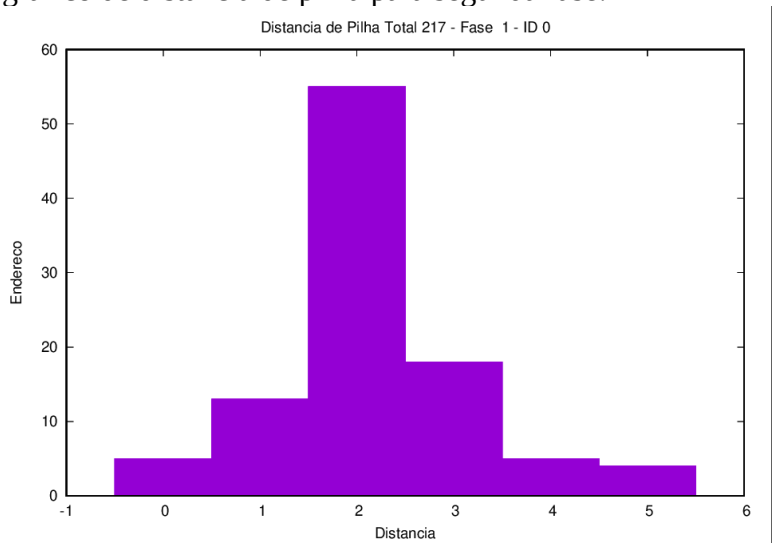
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



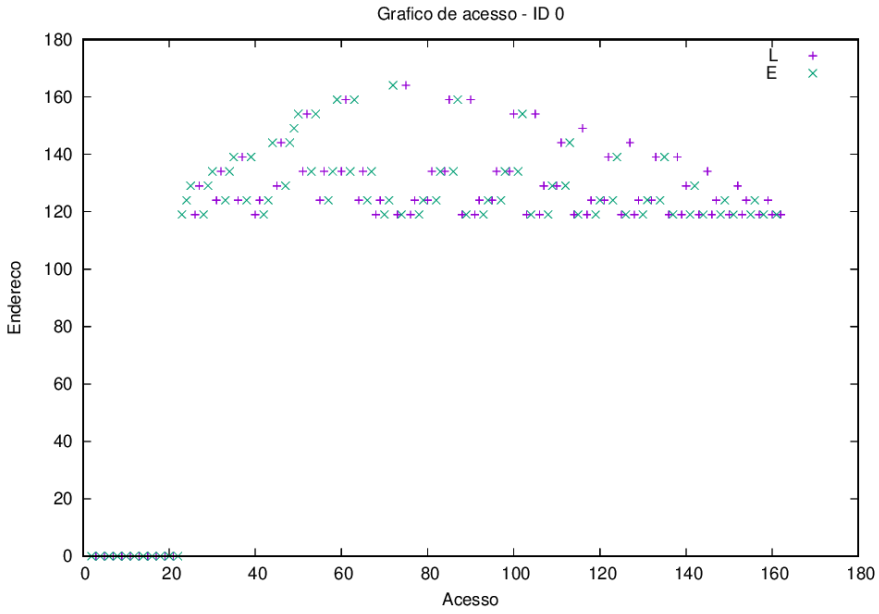
- Exemplo de gráfico de distância de pilha para primeira fase para comparações entre os elementos:



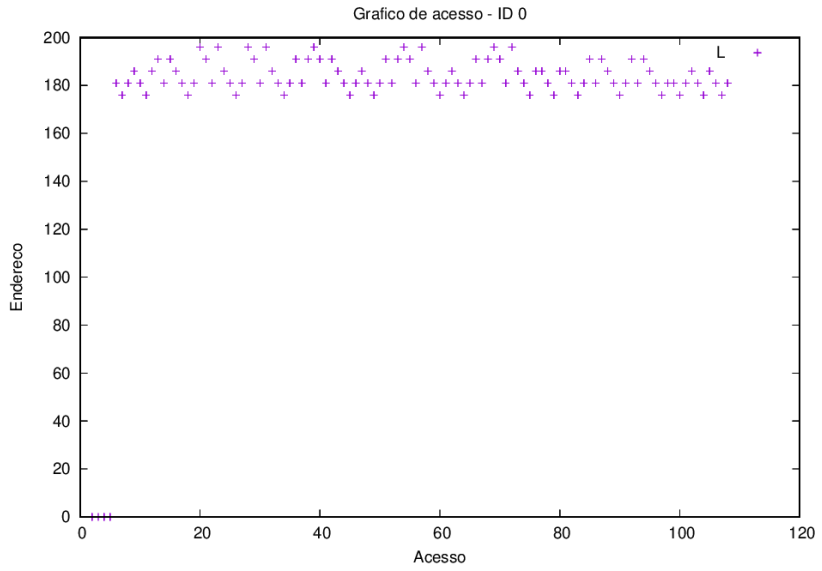
- Exemplo de gráfico de distância de pilha para segunda fase:



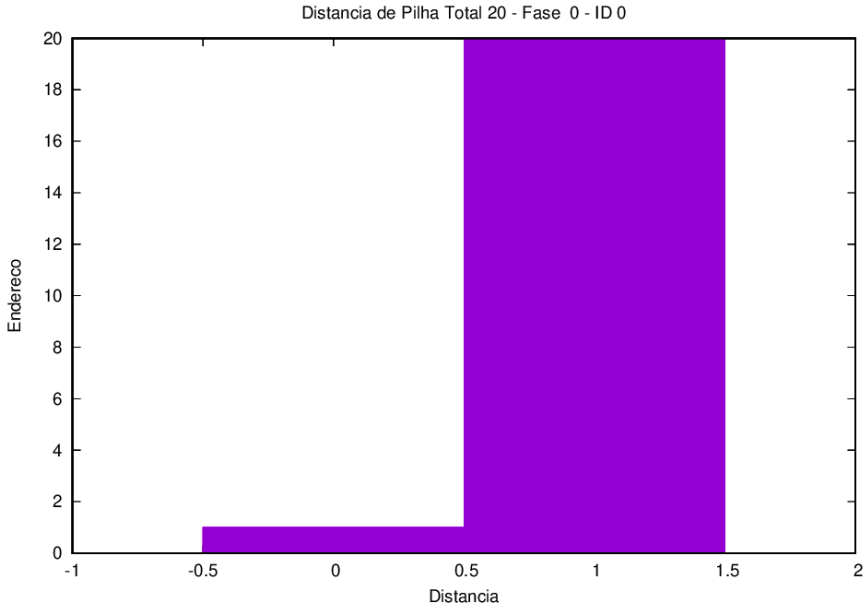
- 10 URLs e 10 rodadas:
  - Tempo de duração médio: 0.0003891046000717324 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



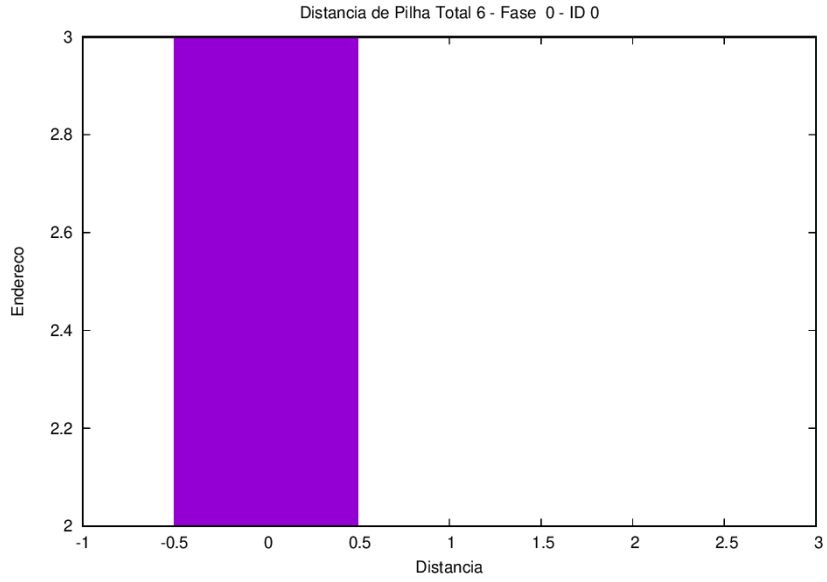
- Exemplo de mapa de acesso à memória para comparações entre os elementos:



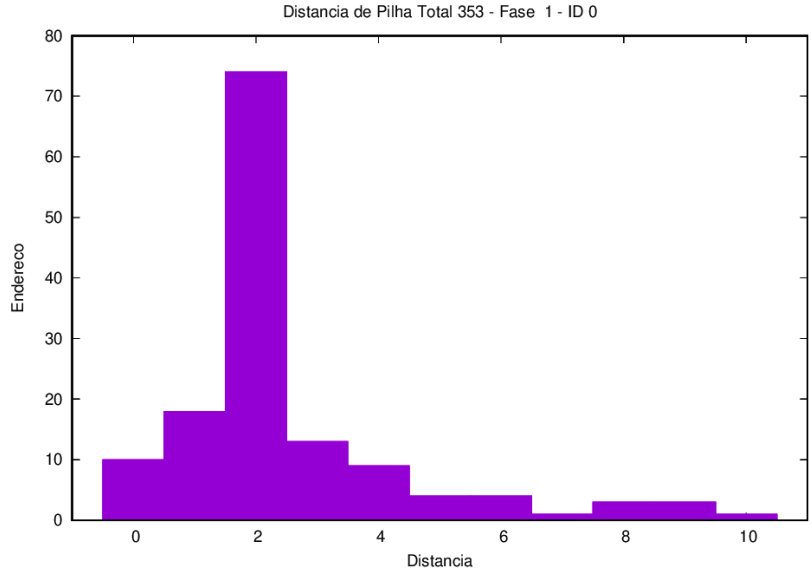
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



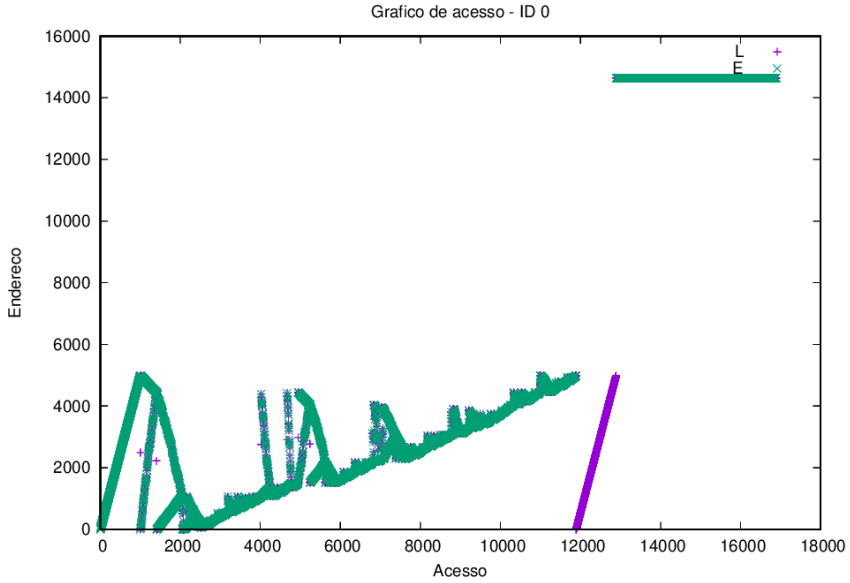
- Exemplo de gráfico de distância de pilha para primeira fase para comparações entre os elementos:



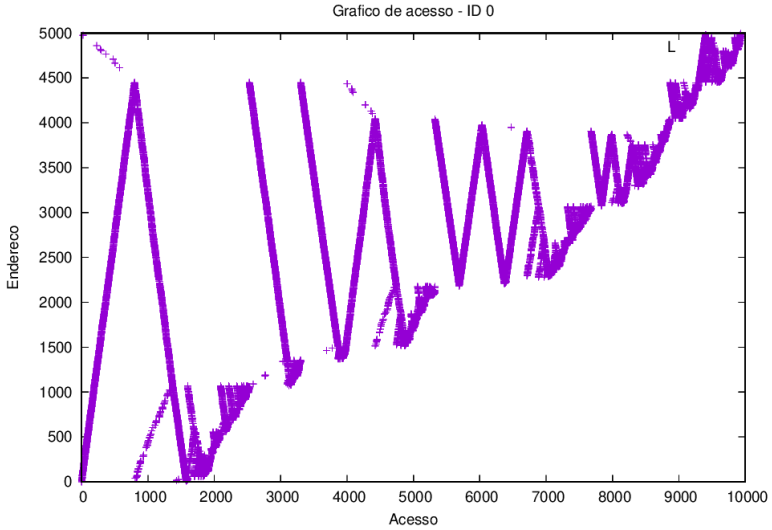
- Exemplo de gráfico de distância de pilha para segunda fase:



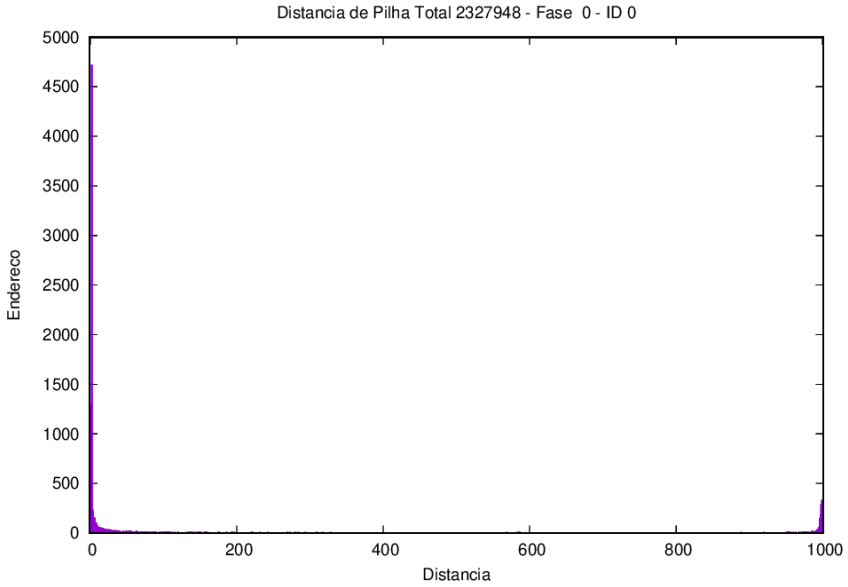
- 1000 URLs e 1 rodada:
  - Tempo de duração médio: 0.005093086099623179 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



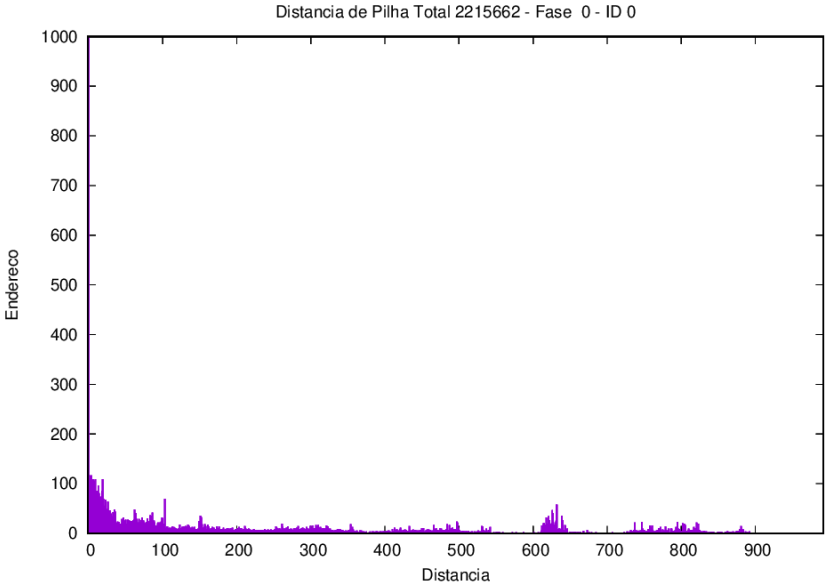
  - Exemplo de mapa de acesso à memória para comparações entre os elementos:



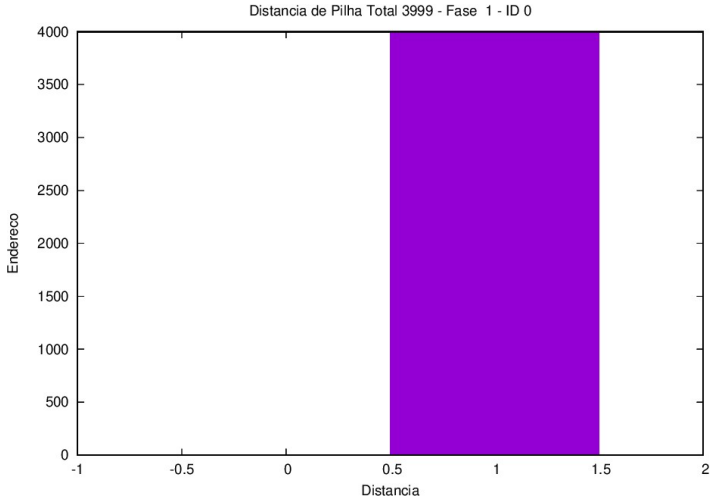
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



- Exemplo de gráfico de distância de pilha para primeira fase para comparações entre os elementos:

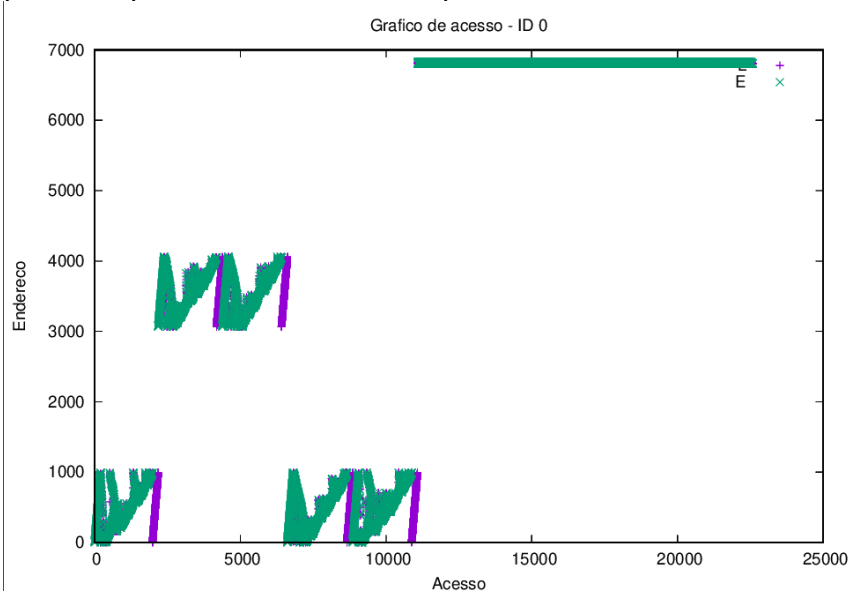


- Exemplo de gráfico de distância de pilha para segunda fase:

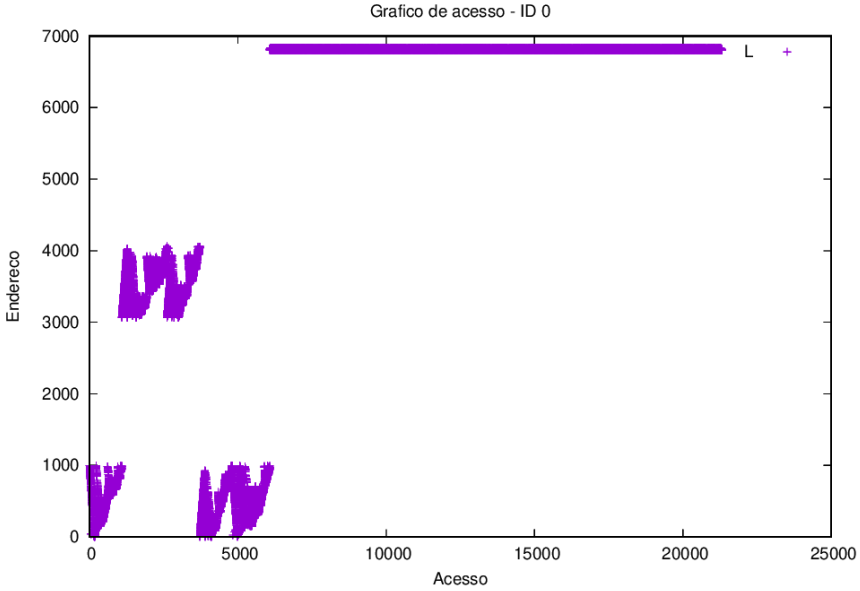


- 1000 URLs e 5 rodadas:
  - Tempo de duração médio: 0.00604639515040617 segundos

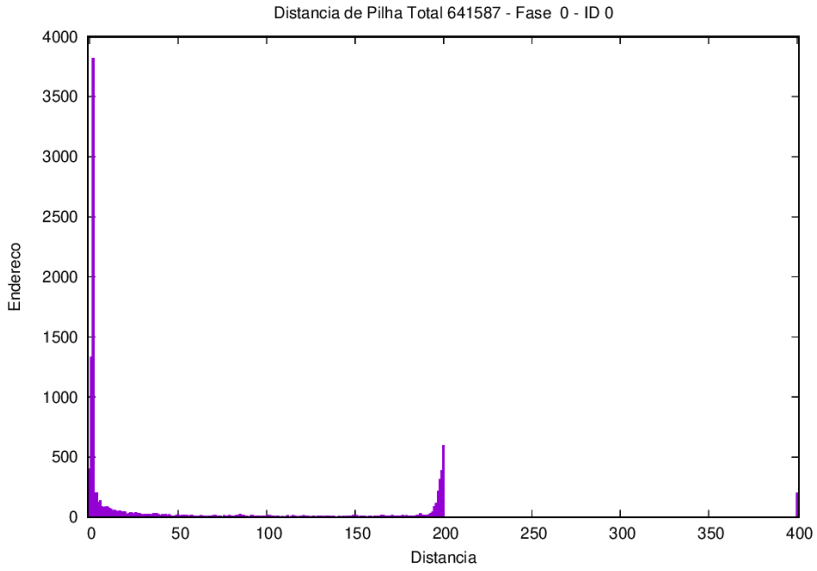
- Exemplo de mapa de acesso à memória para acessos aos elementos:



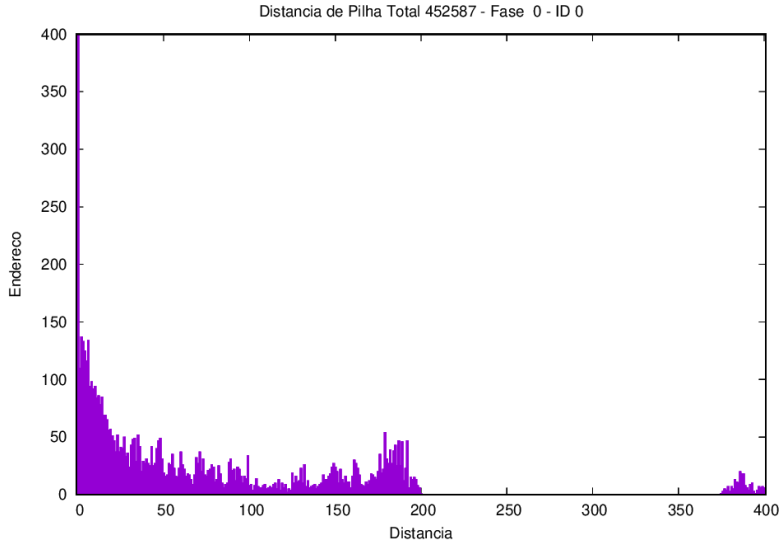
- Exemplo de mapa de acesso à memória para comparações entre os elementos:



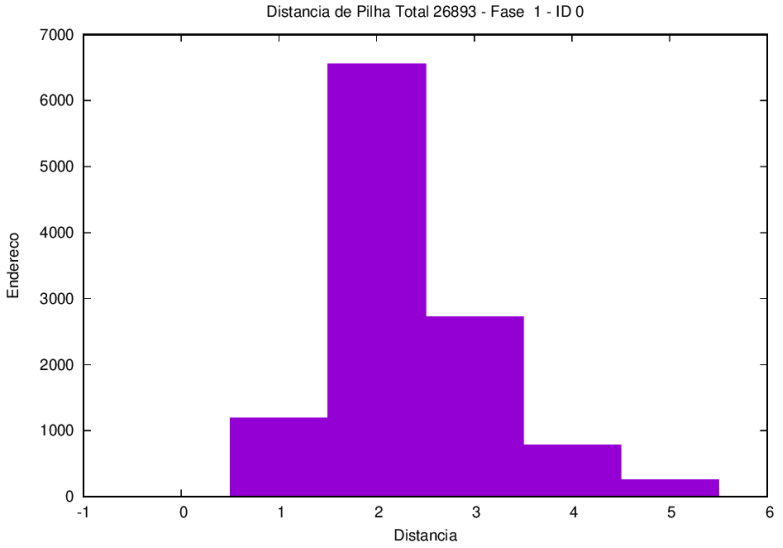
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



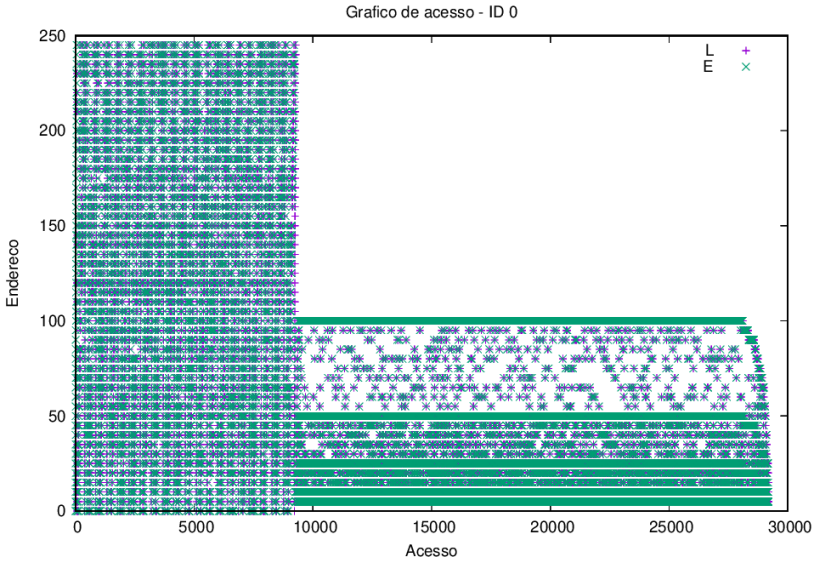
- Exemplo de gráfico de distância de pilha para primeira fase para comparações entre os elementos:



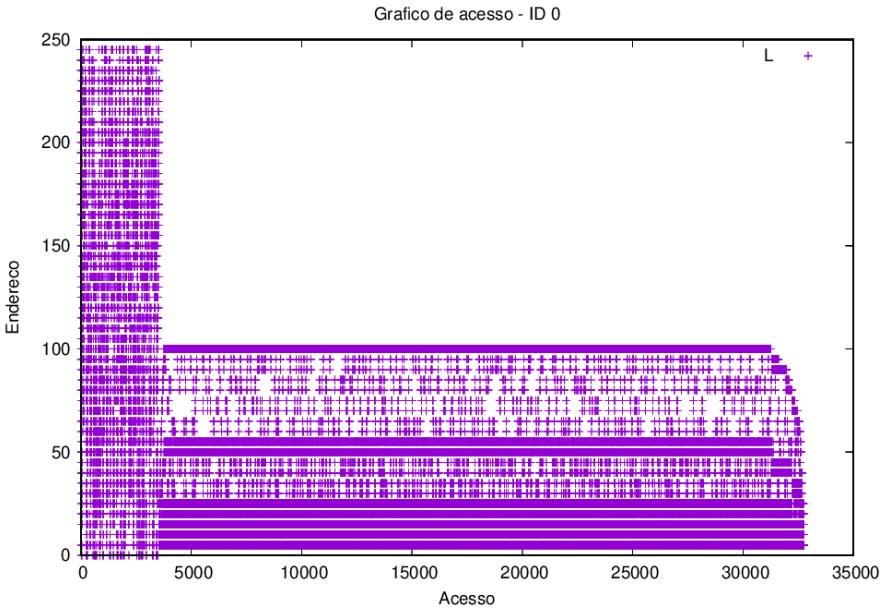
- Exemplo de gráfico de distância de pilha para segunda fase:



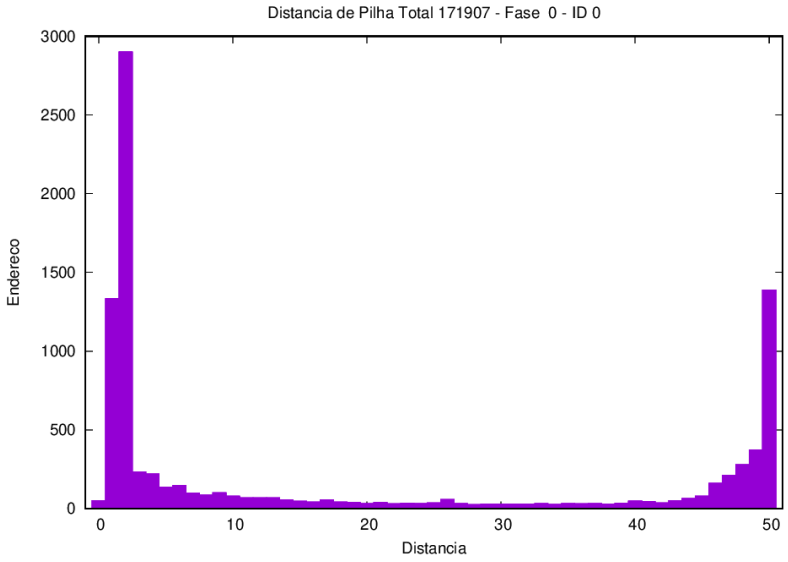
- 1000 URLs e 20 rodadas:
  - Tempo de duração médio: 0.0066025322499626785 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



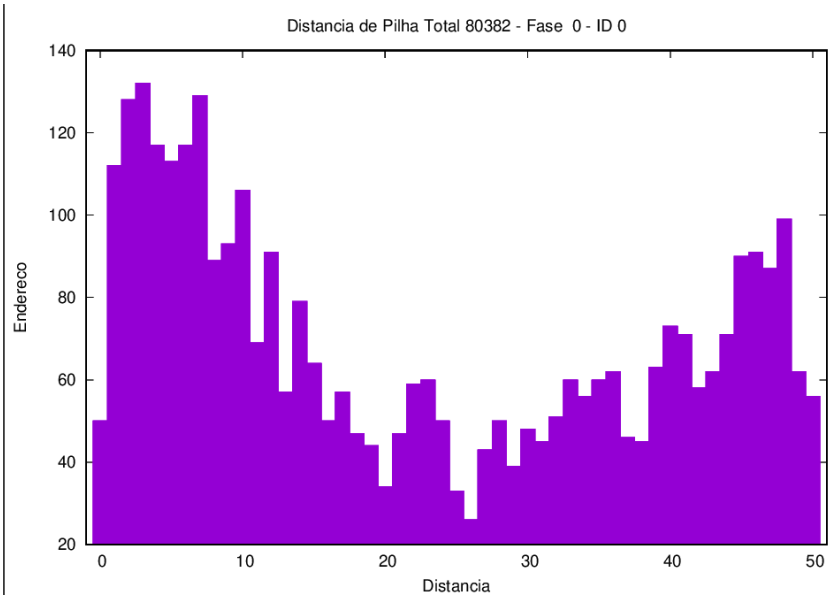
- Exemplo de mapa de acesso à memória para comparações entre os elementos:



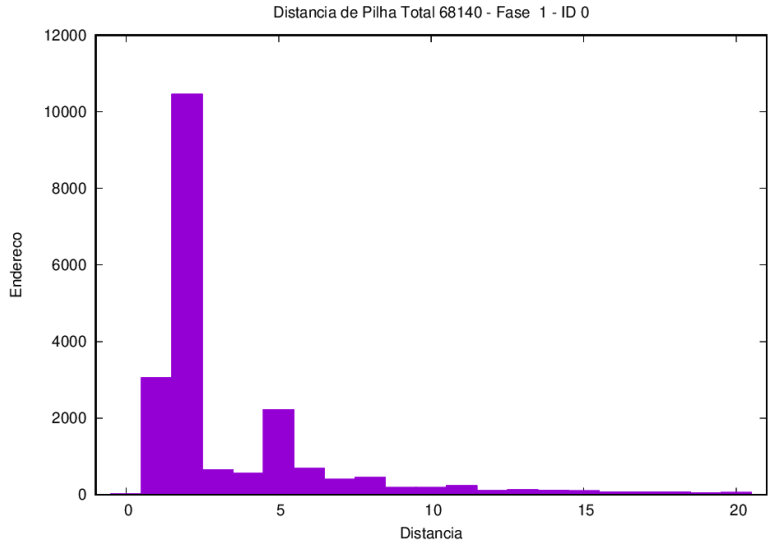
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



- Exemplo de gráfico de distância de pilha para primeira fase para comparações entre os elementos:



- Exemplo de gráfico de distância de pilha para segunda fase:



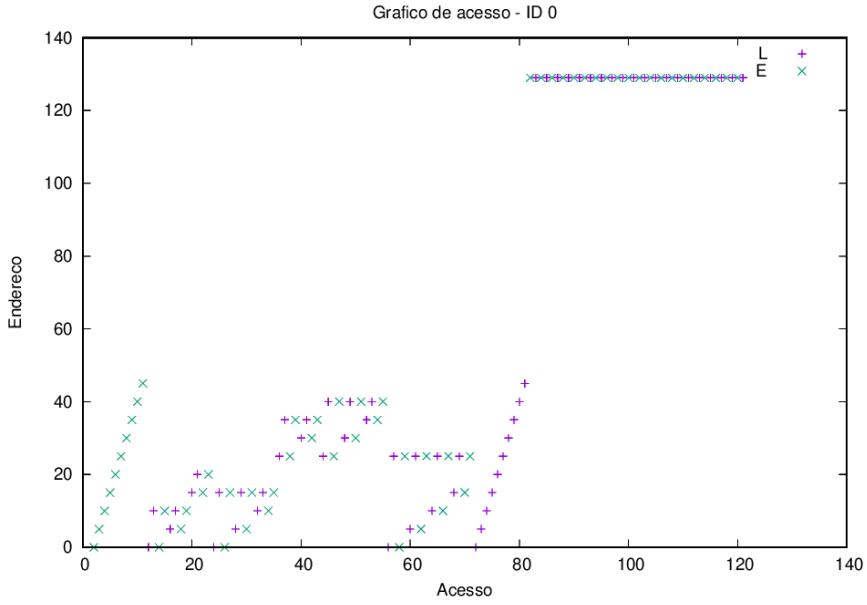
- 100000 URLs e 1 rodada:
  - Tempo de duração médio: 0.534358376200089 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

- 100000 URLs e 5 rodadas:
  - Tempo de duração médio: 0.5897787191502175segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

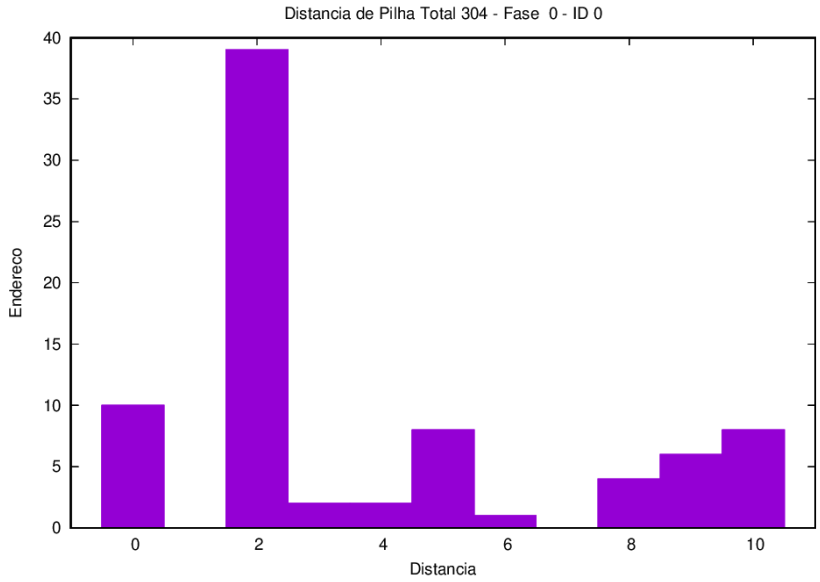
- 100000 URLs e 20 rodadas:
  - Tempo de duração médio: 0.6999017364000792 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

Testes realizados com mergesort implementado recursivamente (desafio 1):

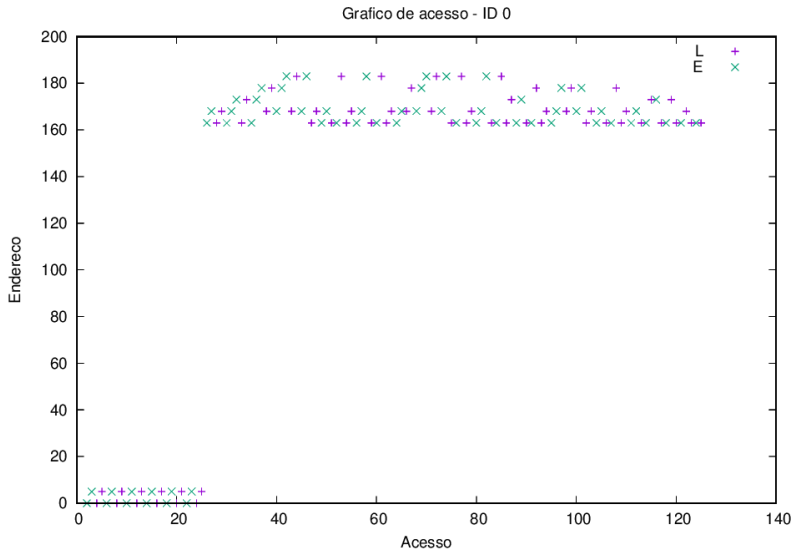
- 10 URLs e 1 rodada:
  - Tempo de duração médio: 0.012048710550152464 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



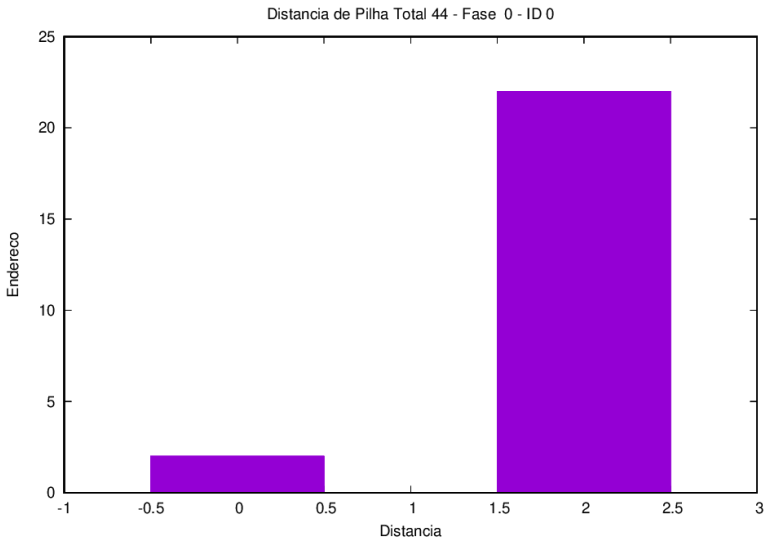
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



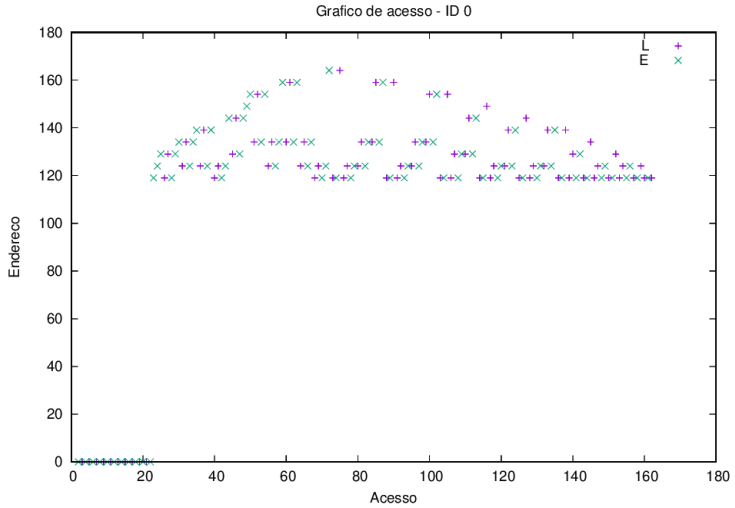
- 10 URLs e 5 rodadas:
  - Tempo de duração médio: 0.00026353580024078837 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



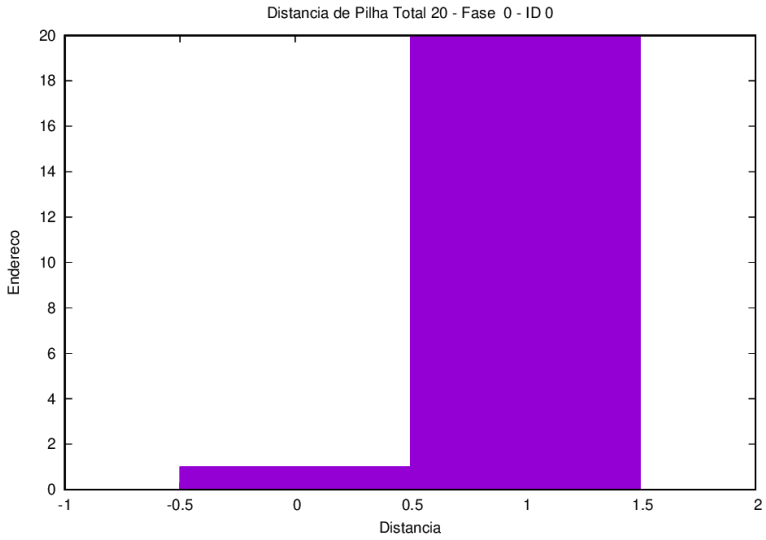
  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



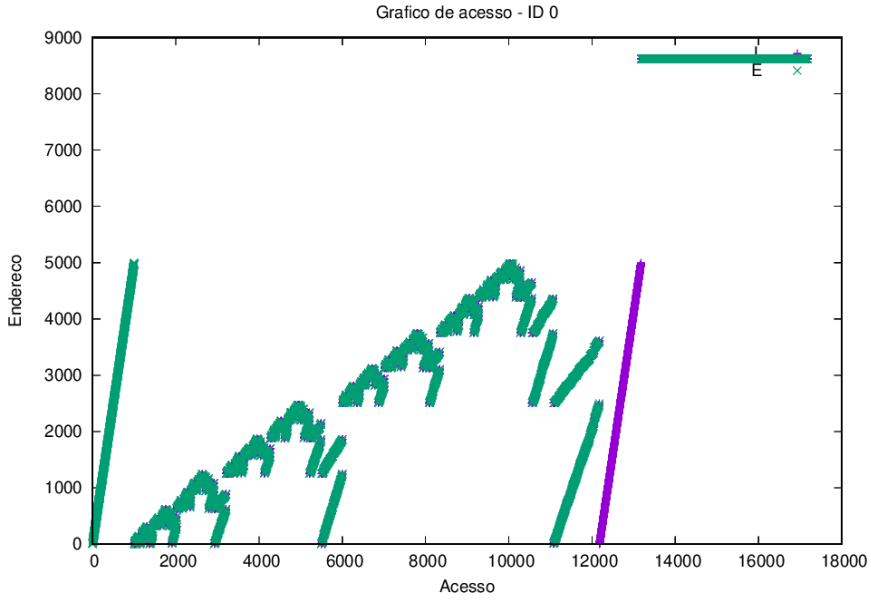
- 10 URLs e 10 rodadas:
  - Tempo de duração médio: 0.0004958806503964297 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



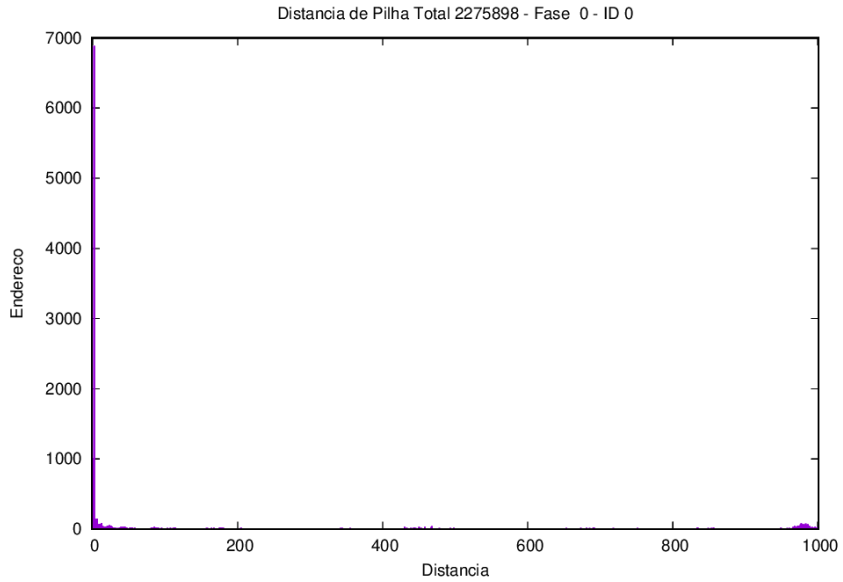
  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



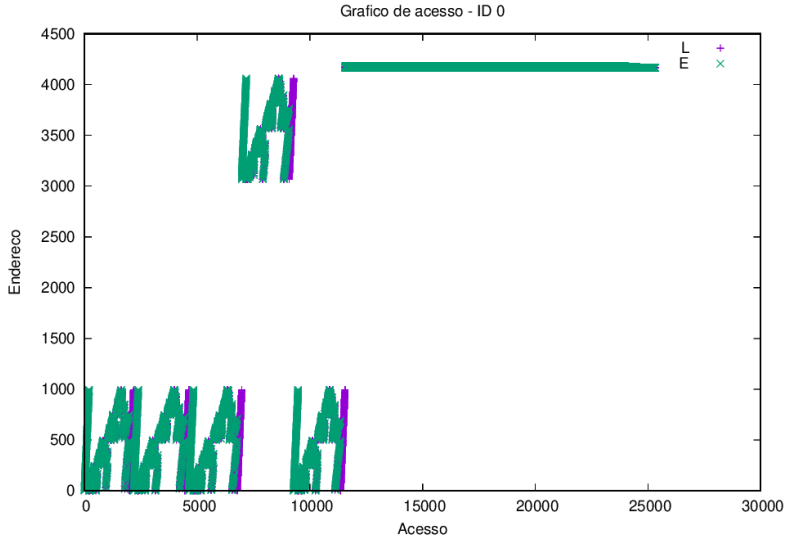
- 1000 URLs e 1 rodada:
  - Tempo de duração médio: 0.004583105449910363 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



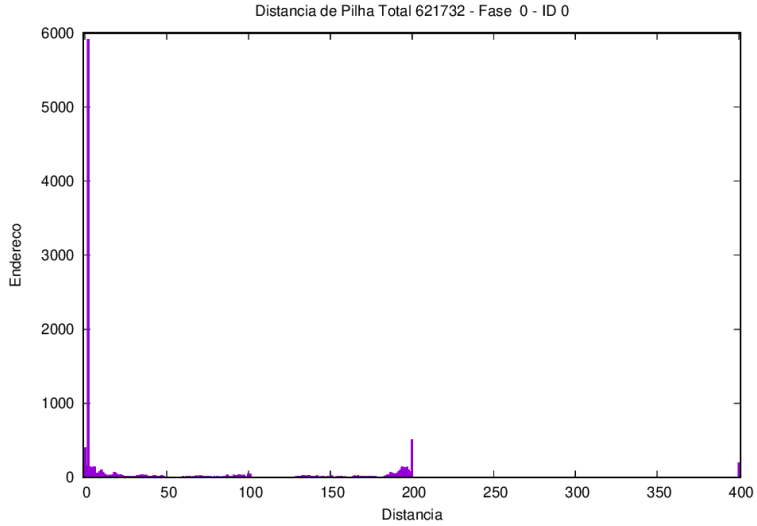
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



- 1000 URLs e 5 rodadas:
  - Tempo de duração médio: 0.0063304947998403804 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:

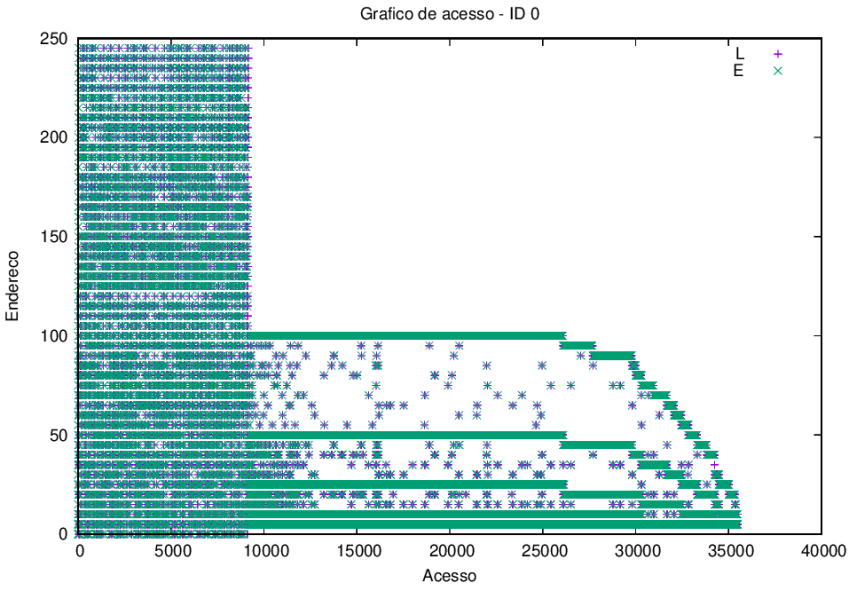


  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:

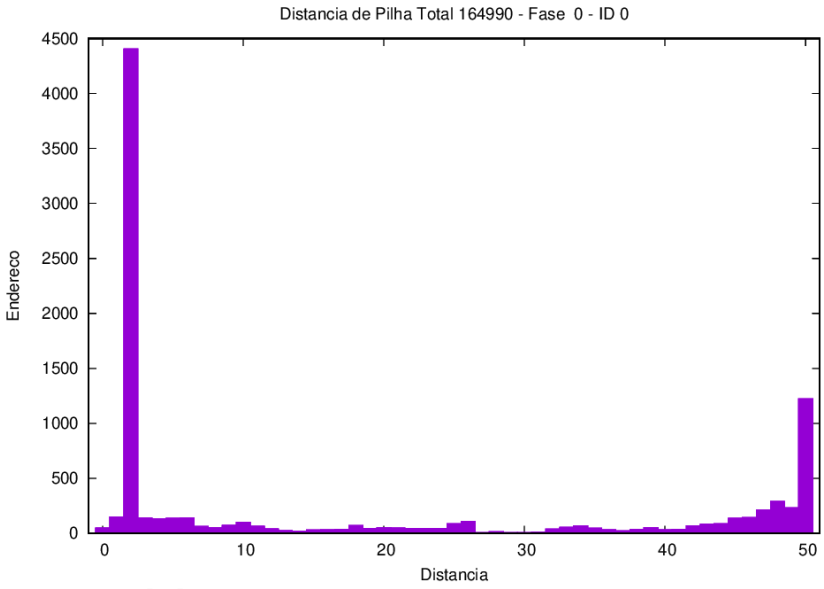


- 1000 URLs e 20 rodadas:
  - Tempo de duração médio: 0.007105982600023708 segundos

- Exemplo de mapa de acesso à memória para acessos aos elementos:



- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



- 100000 URLs e 1 rodada:
  - Tempo de duração médio: 0.4646324438001102 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

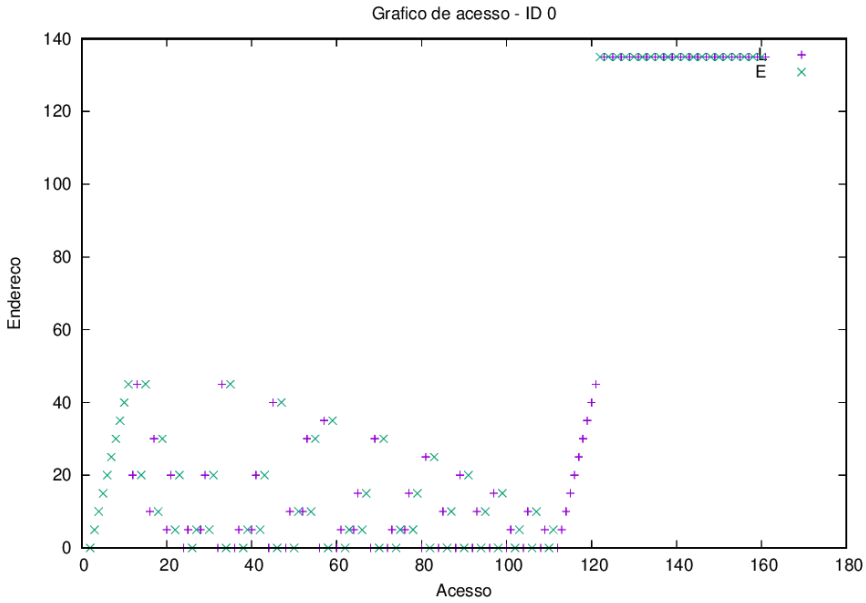
- 100000 URLs e 5 rodadas:
  - Tempo de duração médio: 0.5468357312000081 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

- 100000 URLs e 20 rodadas:
  - Tempo de duração médio: 0.6947513728002377 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

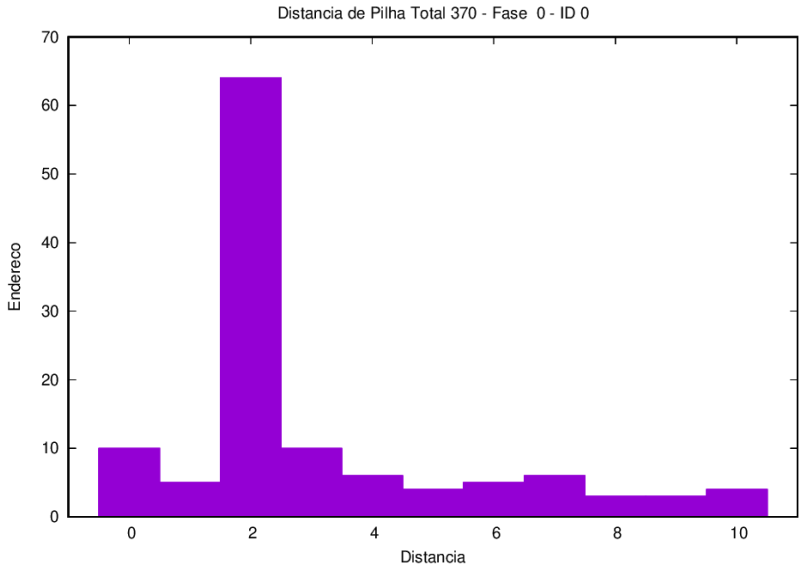
Testes realizados com heapsort (desafio 1):

- 10 URLs e 1 rodada:
  - Tempo de duração médio: 0.0001631130001442216 segundos

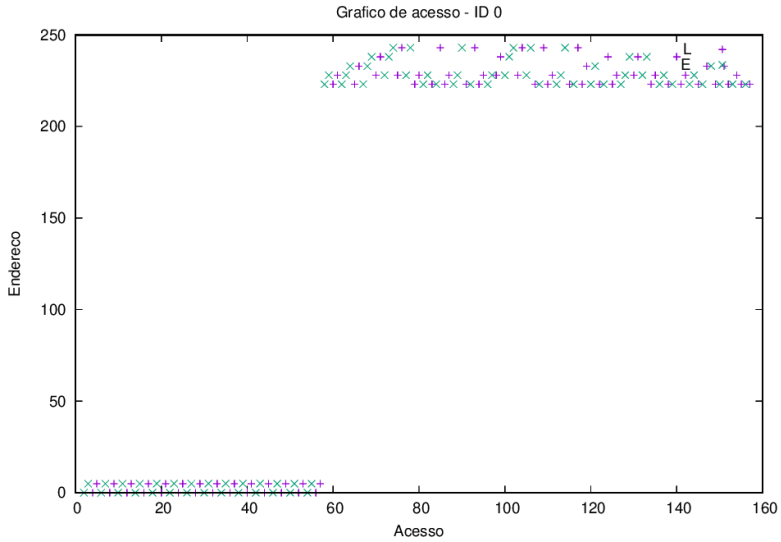
- Exemplo de mapa de acesso à memória para acessos aos elementos:



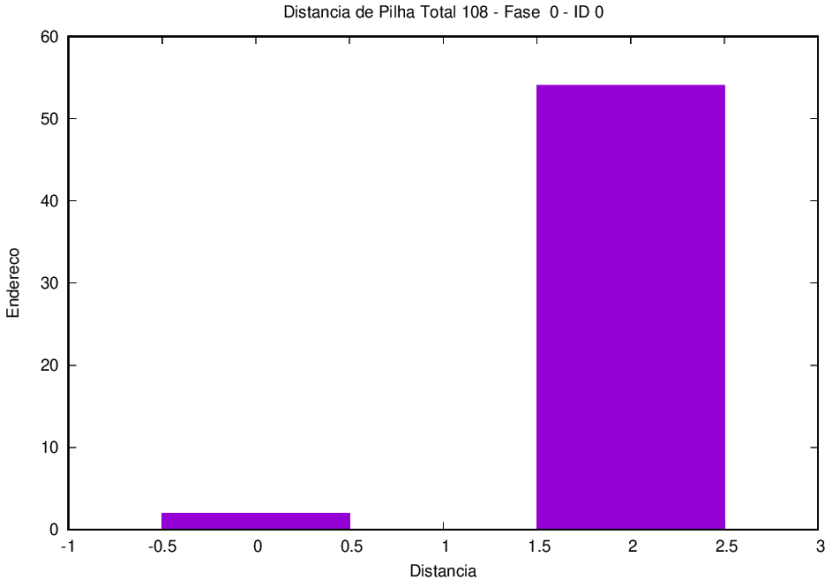
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



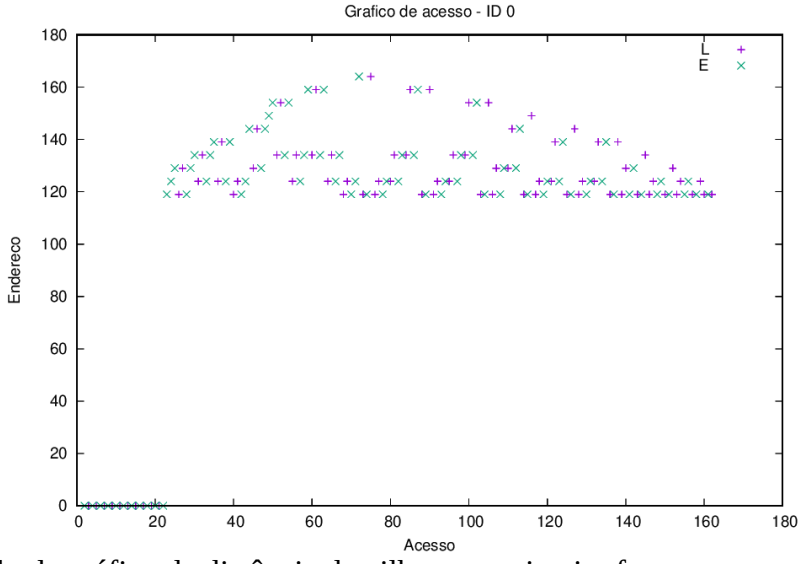
- 10 URLs e 5 rodadas:
  - Tempo de duração médio: 0.0002540870998927858 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



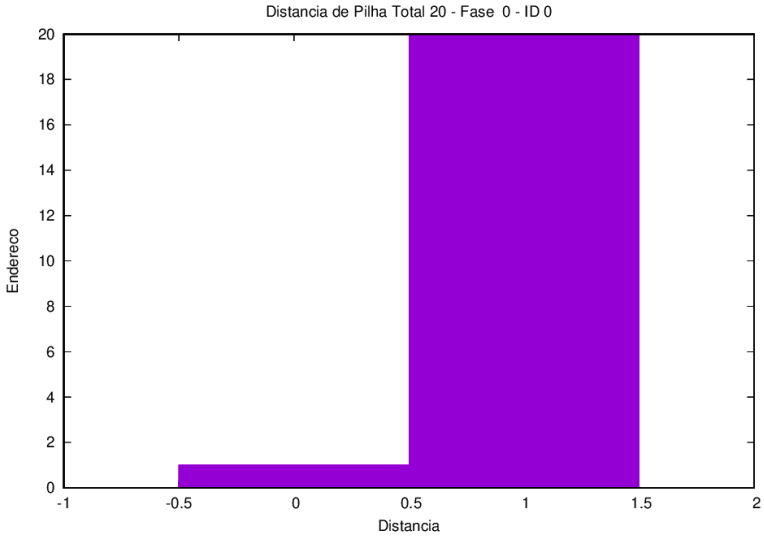
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



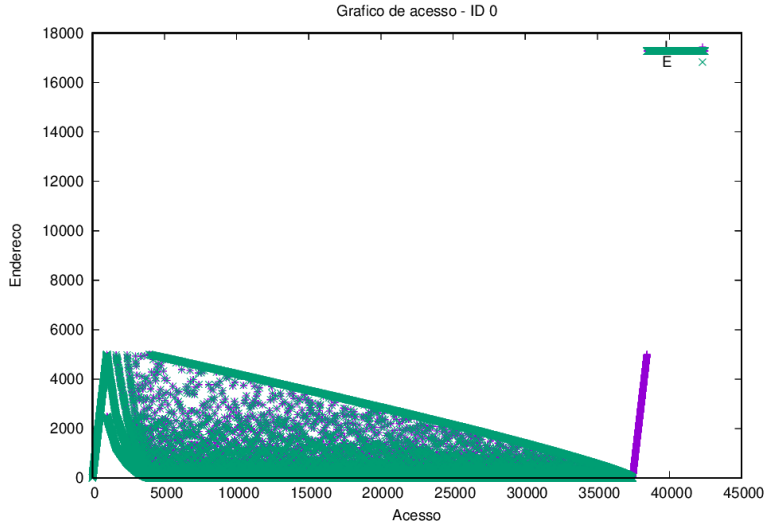
- 10 URLs e 10 rodadas:
  - Tempo de duração médio: 0.0003789588499785168 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



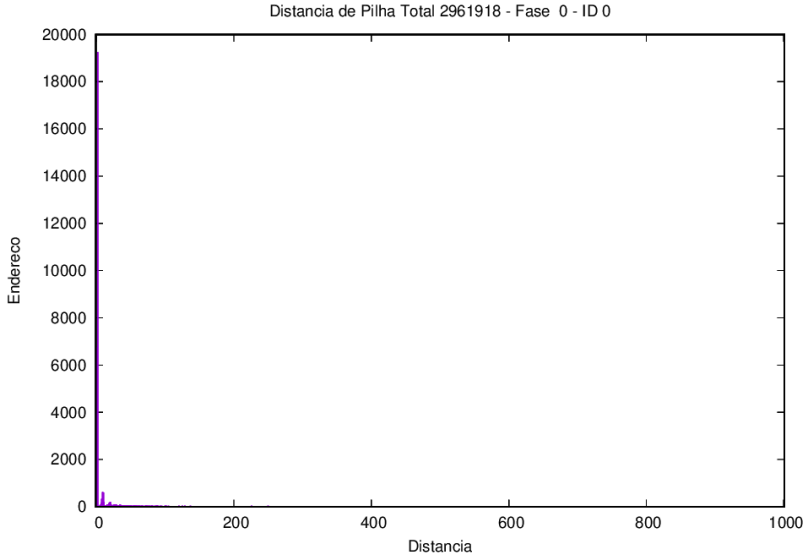
  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



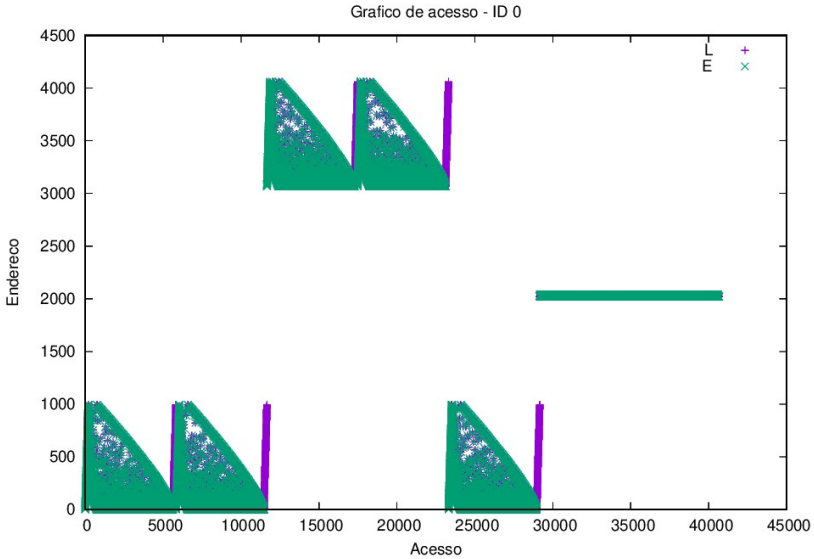
- 1000 URLs e 1 rodada:
  - Tempo de duração médio: 0.006384719799825689 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



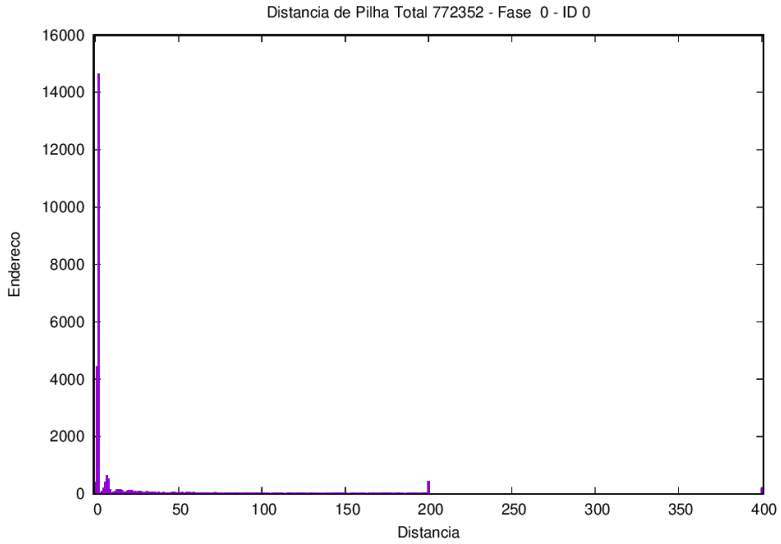
  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



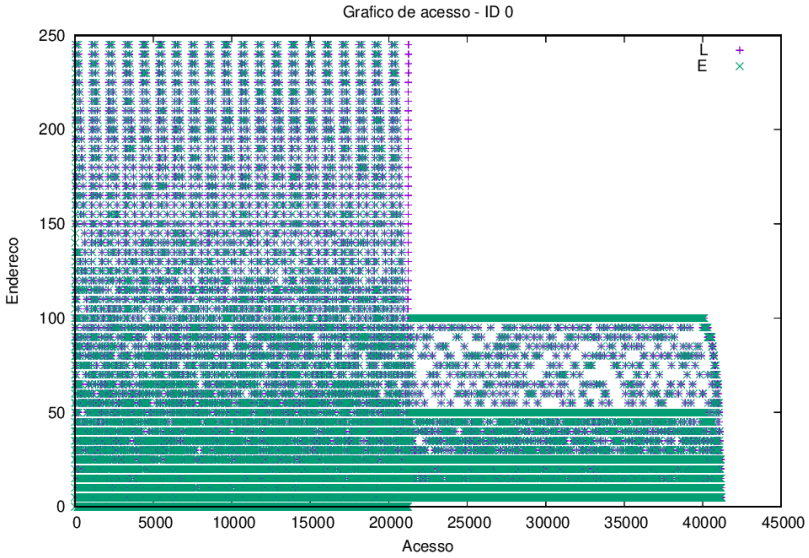
- 1000 URLs e 5 rodadas:
  - Tempo de duração médio: 0.00987396325017471 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



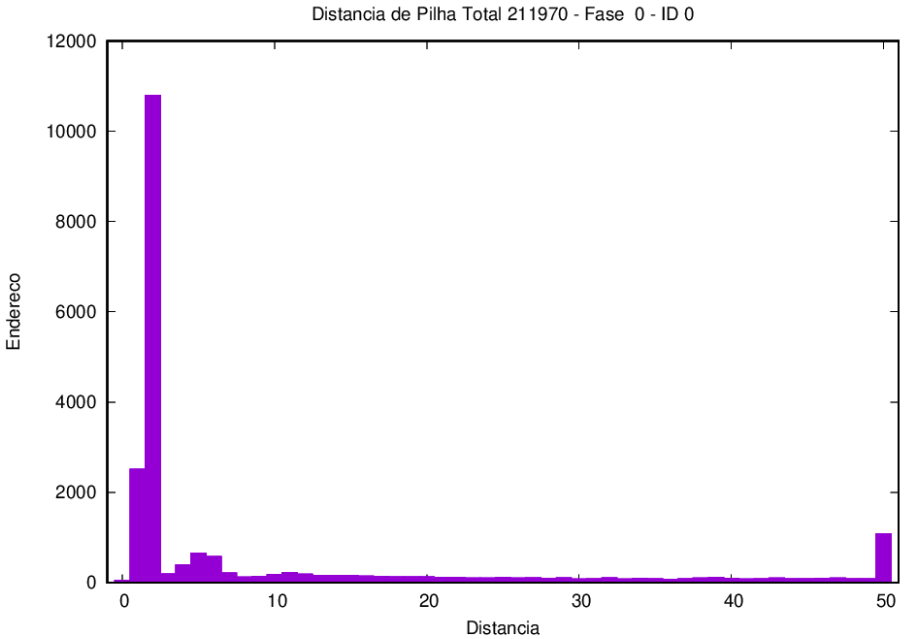
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



- 1000 URLs e 20 rodadas:
  - Tempo de duração médio: 0.0074165532500046535 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



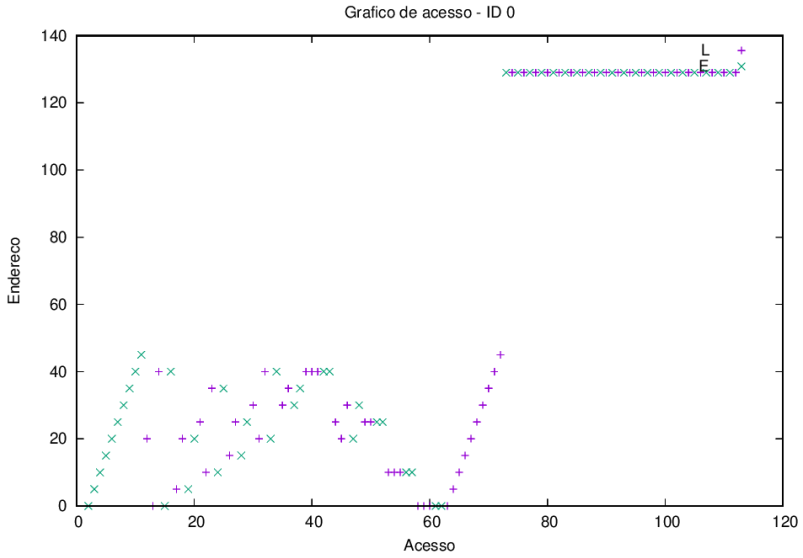
- 100000 URLs e 1 rodada:
  - Tempo de duração médio: 0.8564272081996933 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

- 100000 URLs e 5 rodadas:
  - Tempo de duração médio: 0.8352817665497241 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

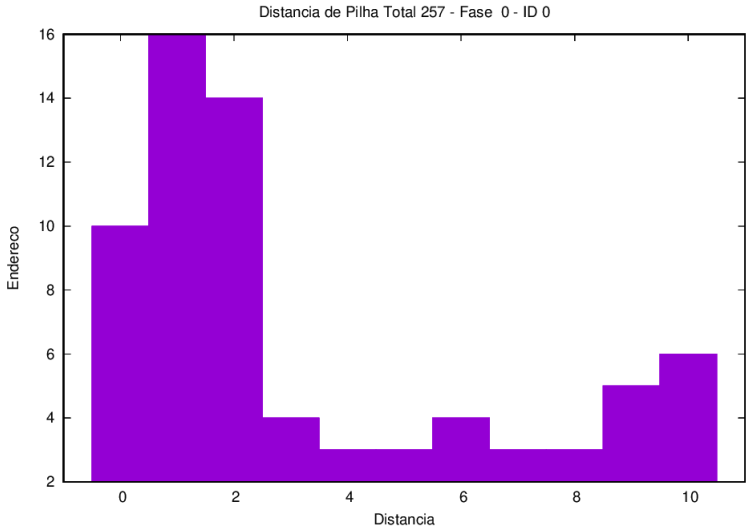
- 100000 URLs e 20 rodadas:
  - Tempo de duração médio: 0.913164708399836 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

Testes realizados com quicksort implementado não-recursivamente (desafio 2):

- 10 URLs e 1 rodada:
  - Tempo de duração médio: 0.0001603159999831405 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:

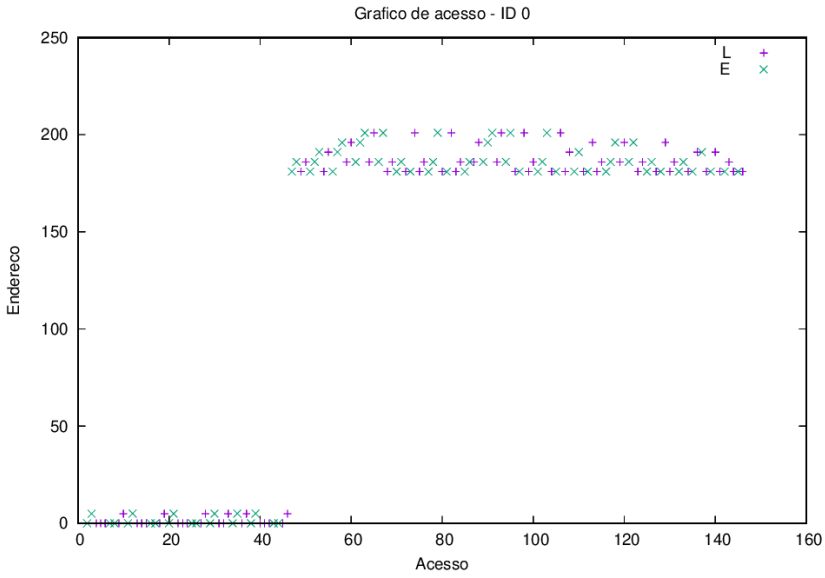


  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:

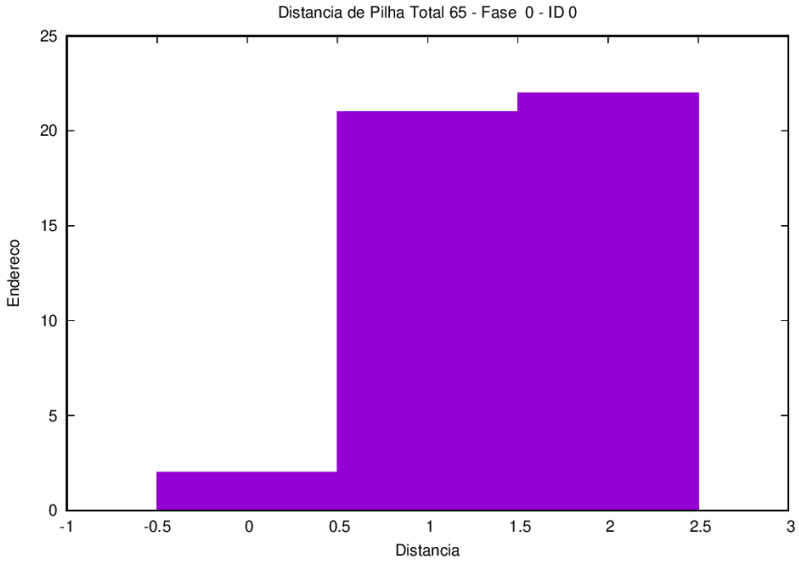


- 10 URLs e 5 rodadas:
  - Tempo de duração médio: 0.0002741496998169168 segundos

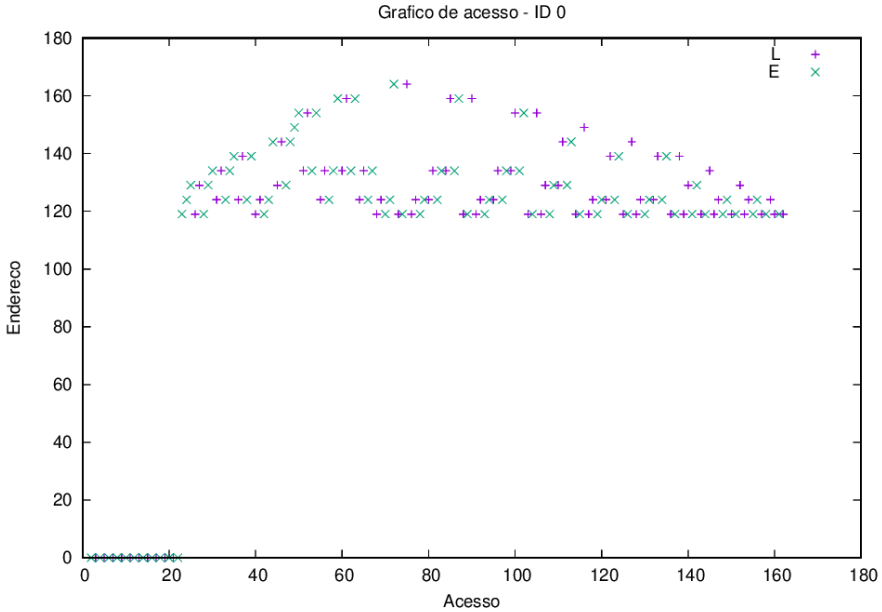
- Exemplo de mapa de acesso à memória para acessos aos elementos:



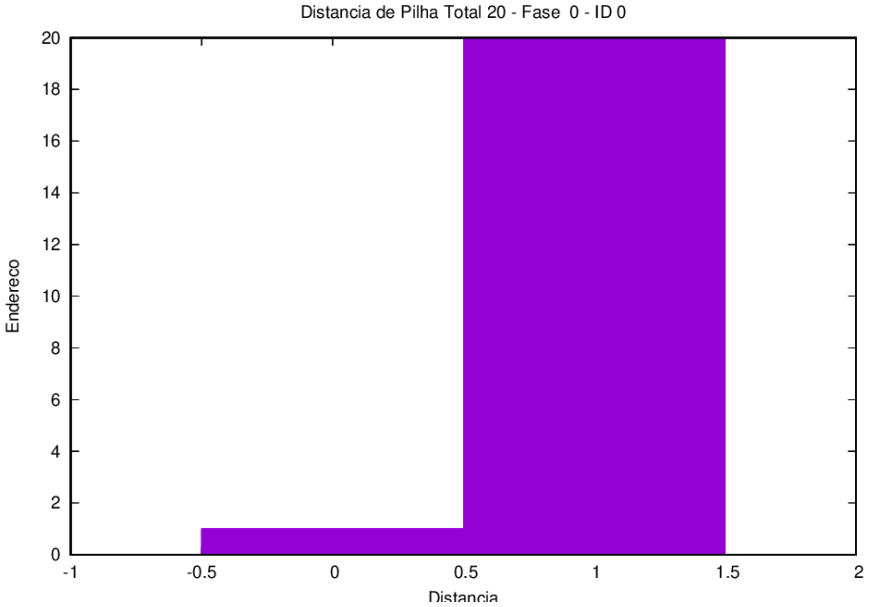
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



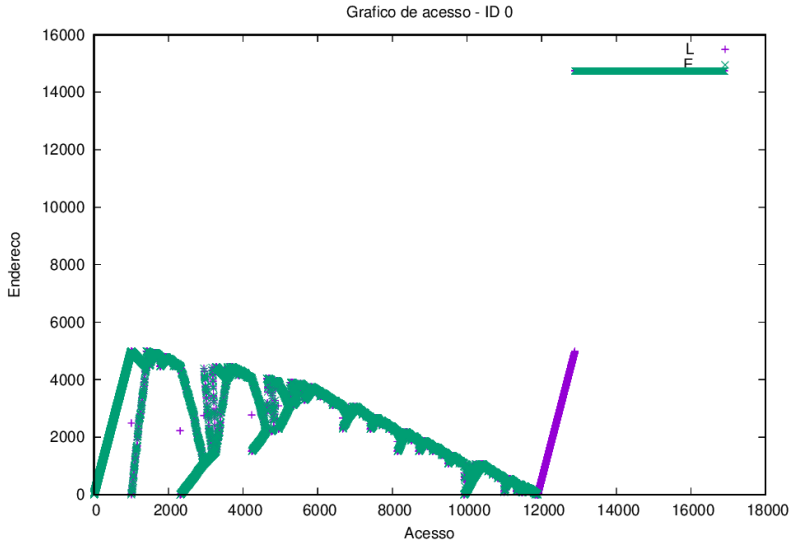
- 10 URLs e 10 rodadas:
  - Tempo de duração médio: 0.000381899500007421 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



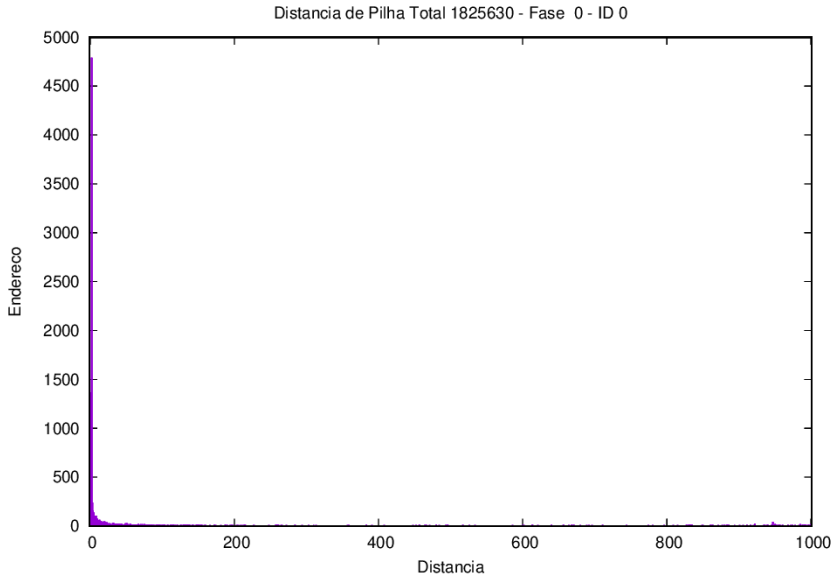
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



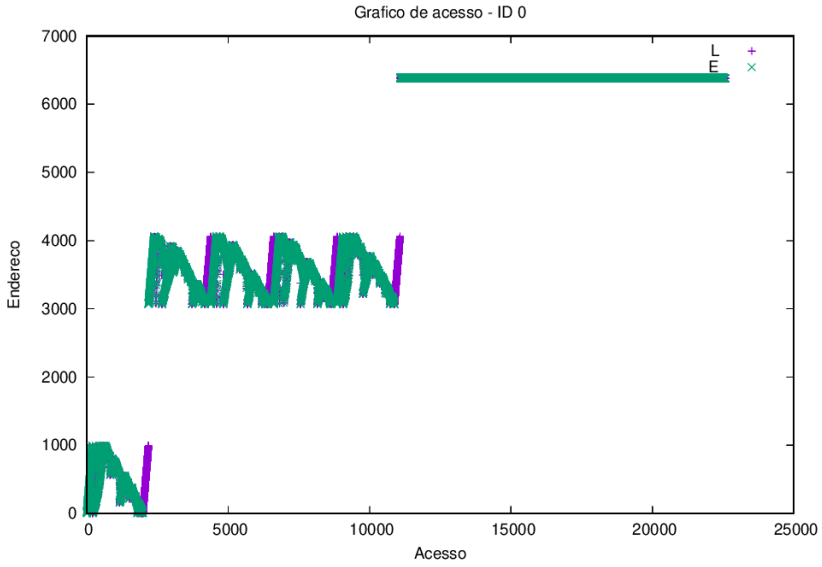
- 1000 URLs e 1 rodada:
  - Tempo de duração médio: 0.005047199949876813 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



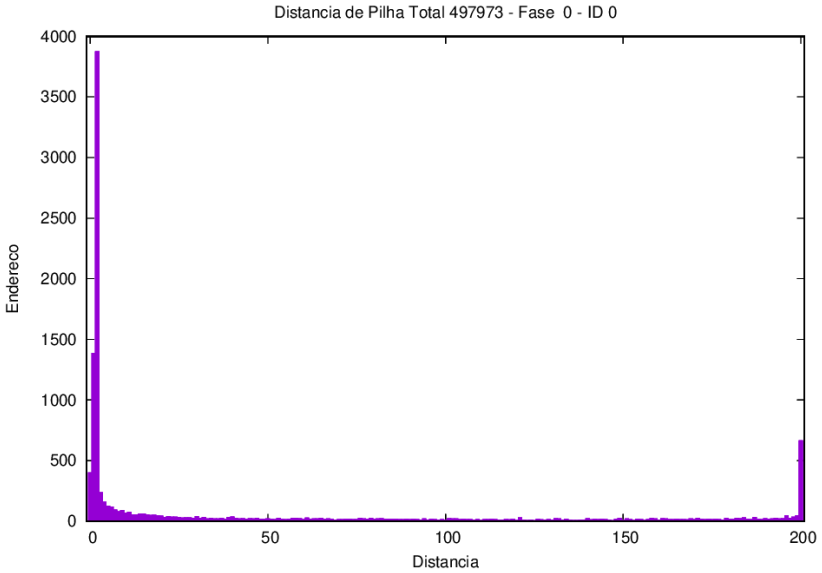
  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



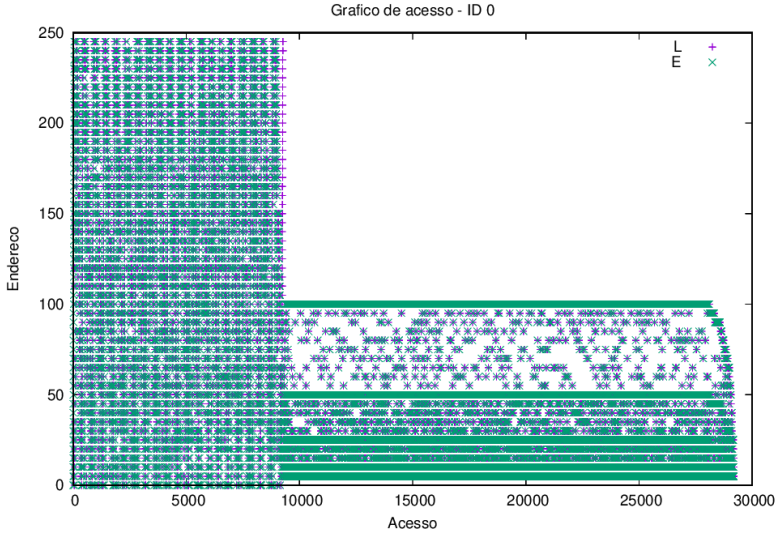
- 1000 URLs e 5 rodadas:
  - Tempo de duração médio: 0.006019343749903783 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



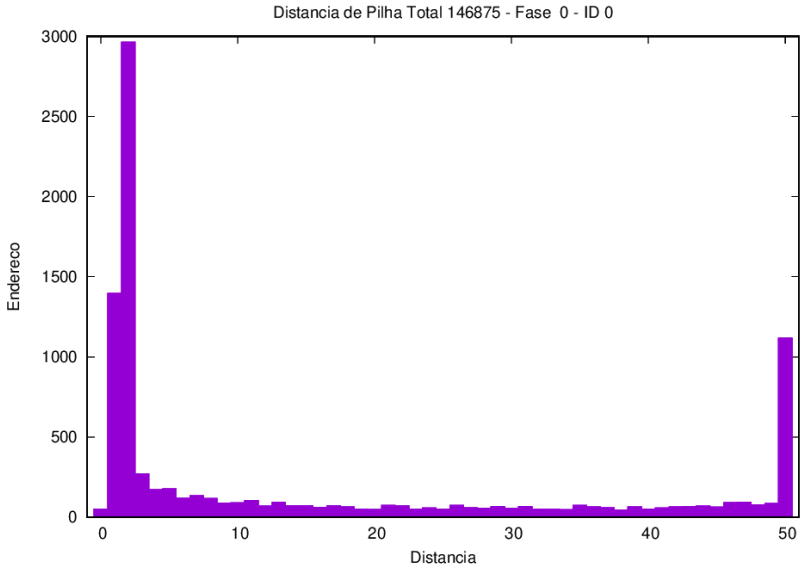
  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



- 1000 URLs e 20 rodadas:
  - Tempo de duração médio: 0.006687184700422222 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



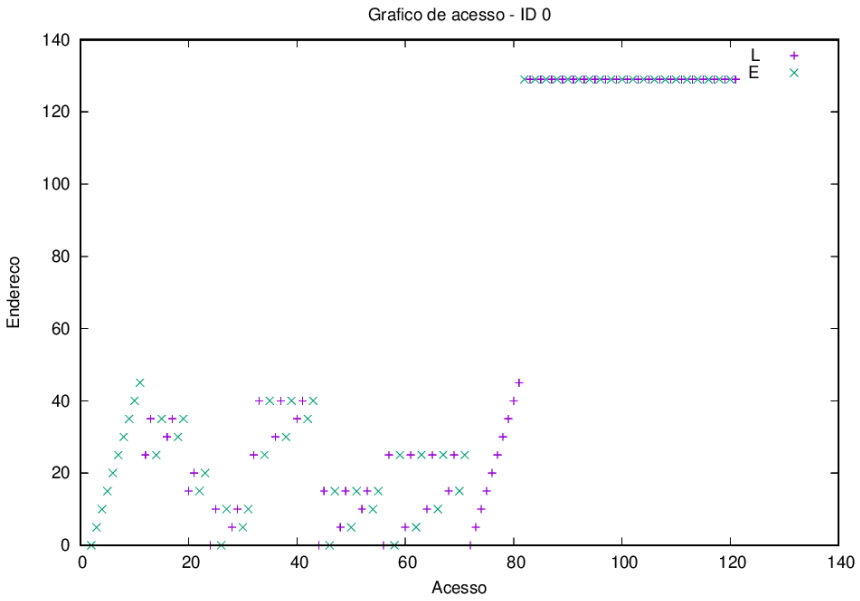
- 100000 URLs e 1 rodada:
  - Tempo de duração médio: 0.5364907371001209 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

- 100000 URLs e 5 rodadas:
  - Tempo de duração médio: 0.5936179872499452 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

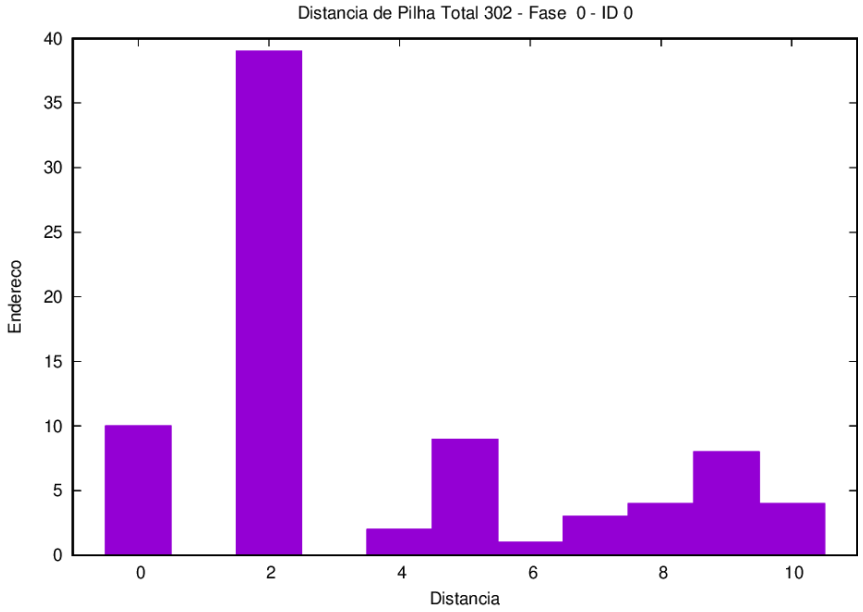
- 100000 URLs e 20 rodadas:
  - Tempo de duração médio: 0.7120126658999653 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

Testes realizados com mergesort implementado não-recursivamente (desafio 2):

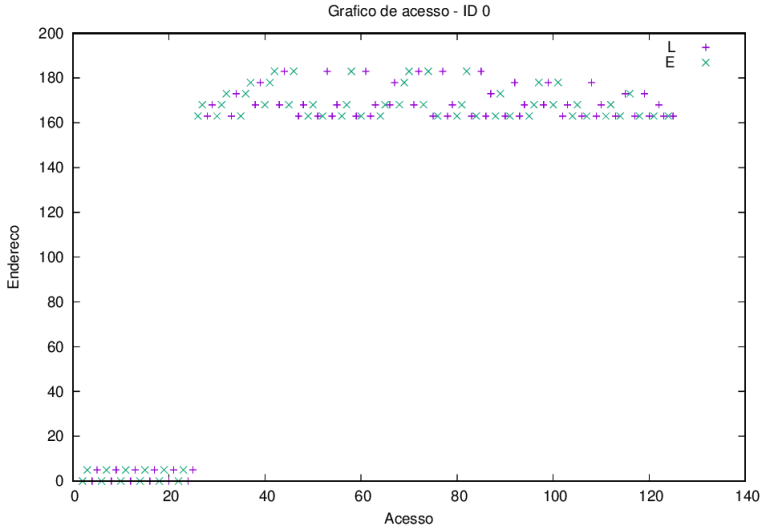
- 10 URLs e 1 rodada:
  - Tempo de duração médio: 0.00015971180009728415 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



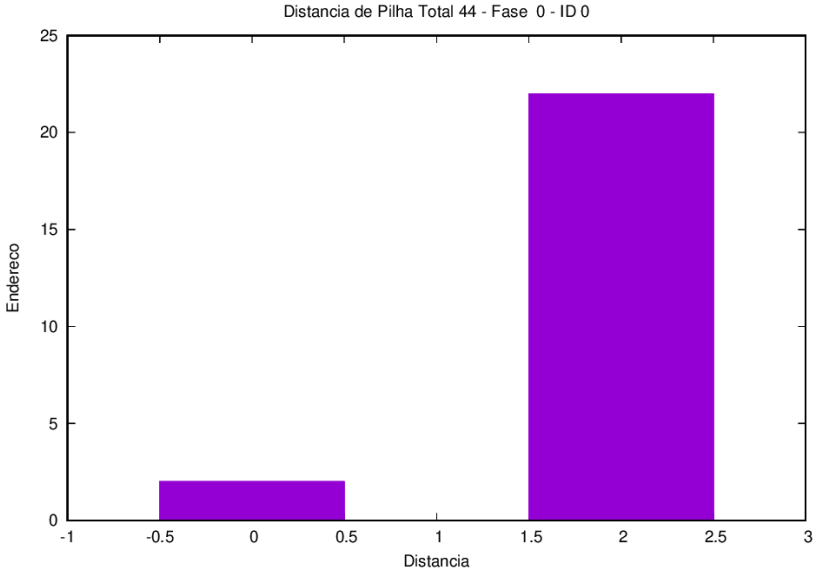
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



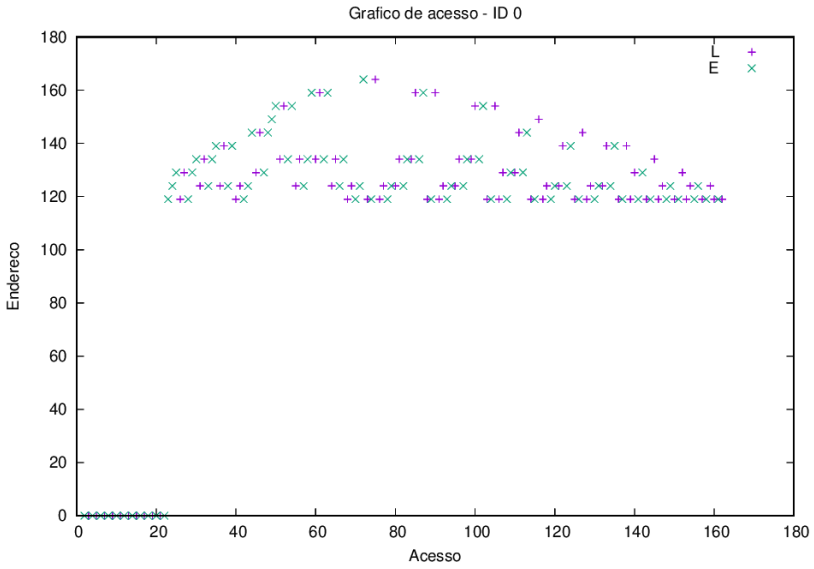
- 10 URLs e 5 rodadas:
  - Tempo de duração médio: 0.0002577148999989731 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



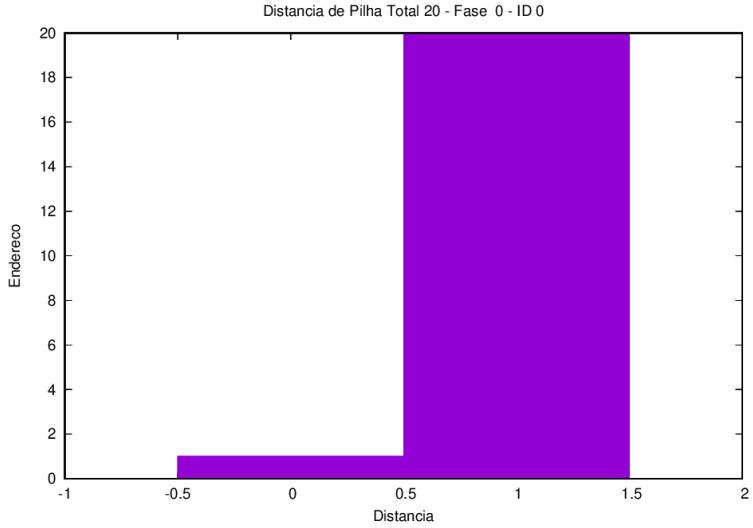
  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



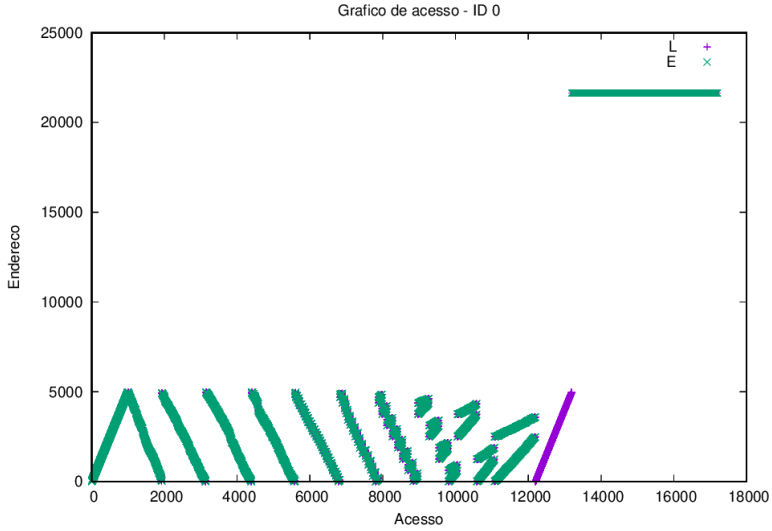
- 10 URLs e 10 rodadas:
  - Tempo de duração médio: 0.0003974469500462874 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



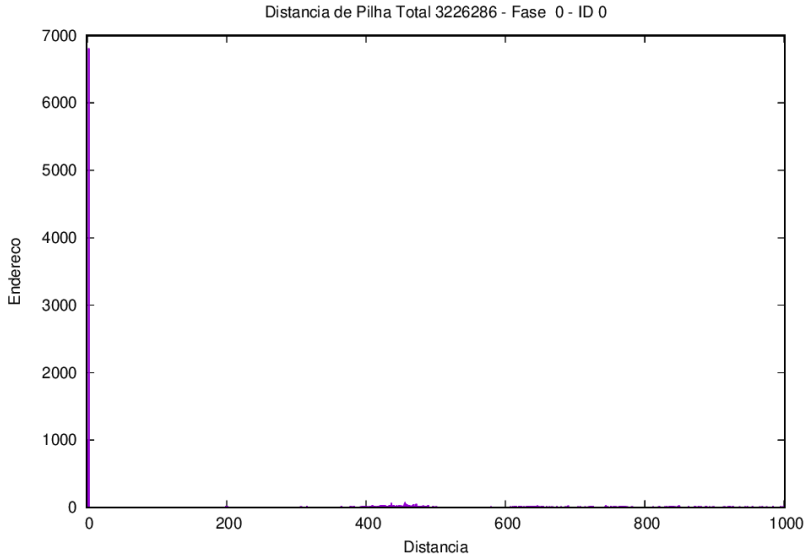
  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



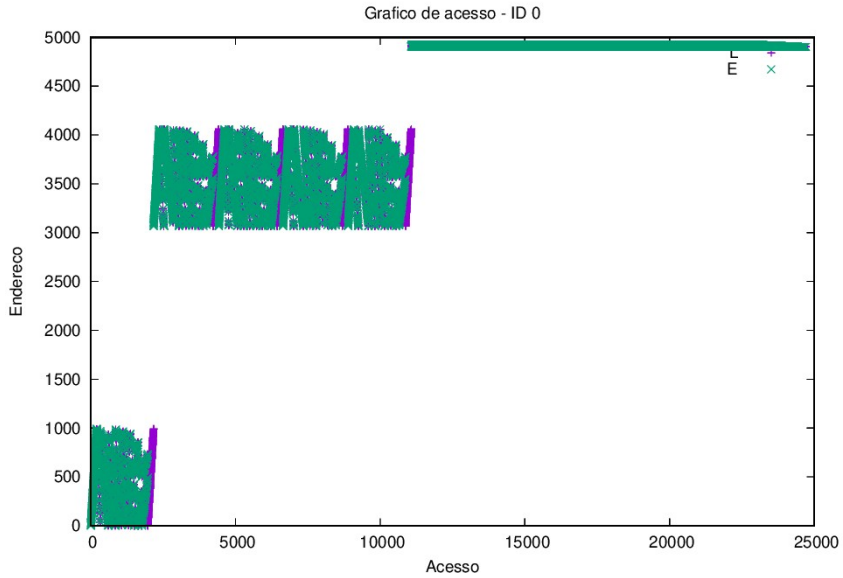
- 1000 URLs e 1 rodada:
  - Tempo de duração médio: 0.004576551599893719 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



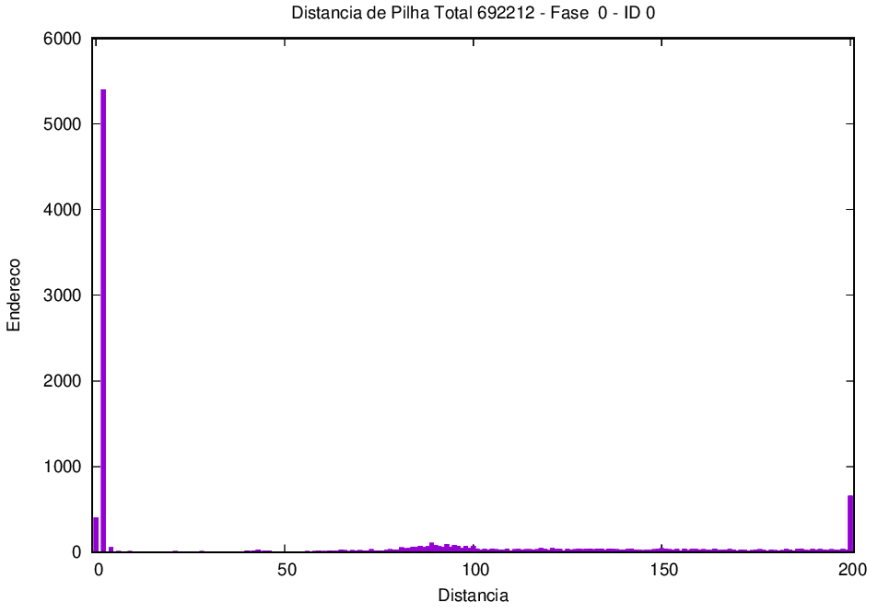
- Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



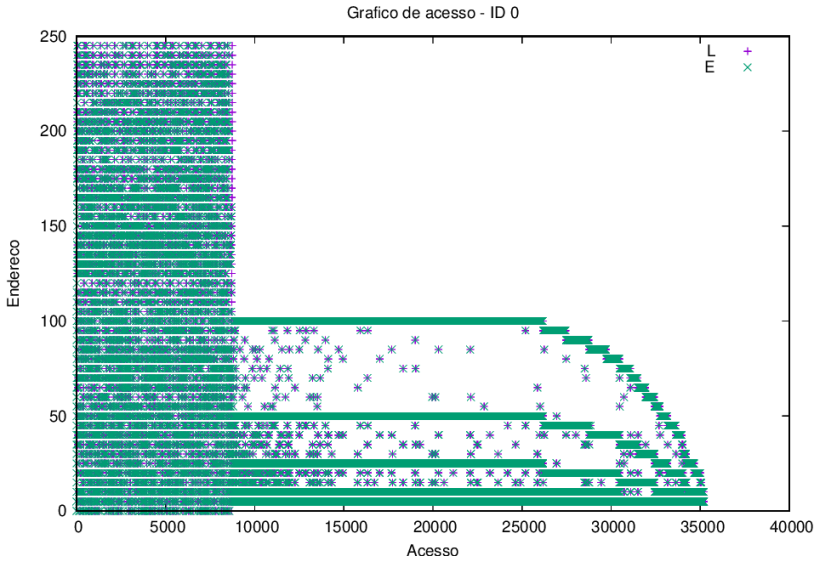
- 1000 URLs e 5 rodadas:
  - Tempo de duração médio: 0.005572429350195307 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



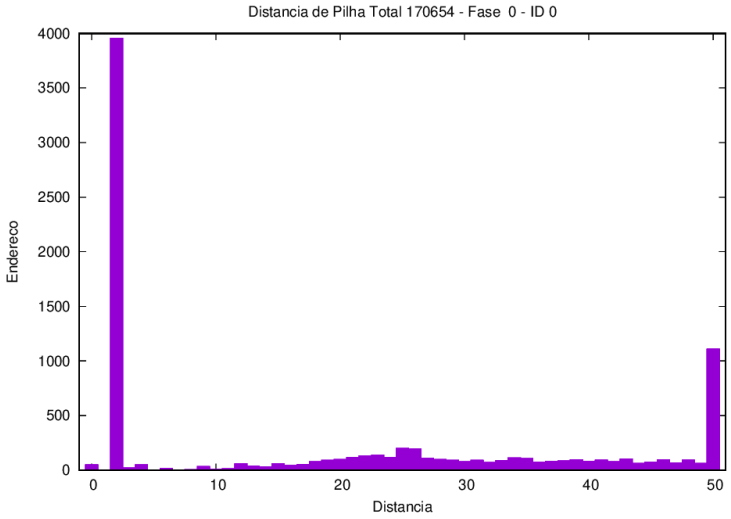
  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



- 1000 URLs e 20 rodadas:
  - Tempo de duração médio: 0.007537982999929227 segundos
  - Exemplo de mapa de acesso à memória para acessos aos elementos:



  - Exemplo de gráfico de distância de pilha para primeira fase para acessos aos elementos:



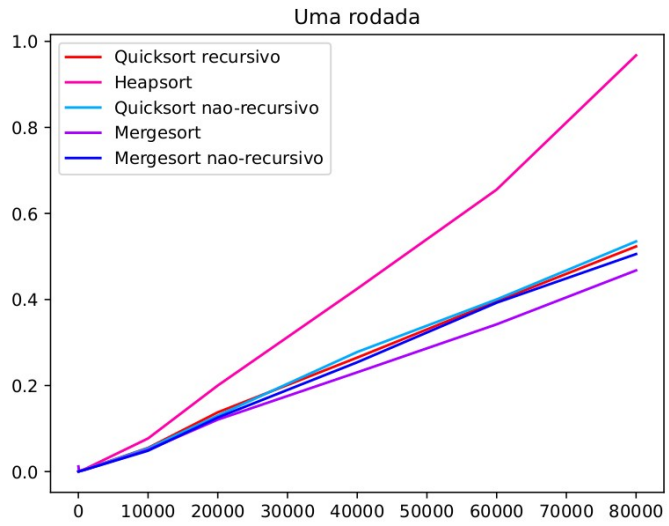
- 100000 URLs e 1 rodada:
  - Tempo de duração médio: 0.5261196832498172 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

- 100000 URLs e 5 rodadas:
  - Tempo de duração médio: 0.5366219093999461 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

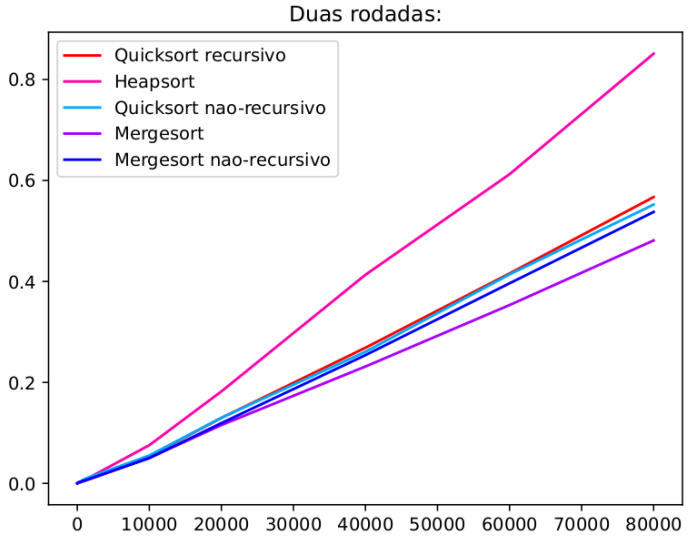
- 100000 URLs e 20 rodadas:
  - Tempo de duração médio: 0.6918661674998475 segundos
  - Exemplo de mapa de acesso à memória não será apresentado por limitações de tempo e espaço para plotagem do mesmo.

Gráfico de tempo de execução em função da quantidade total de URLs (incluindo 10, 100, 1000, 10000, 20000, 40000, 60000 e 80000 URLs):

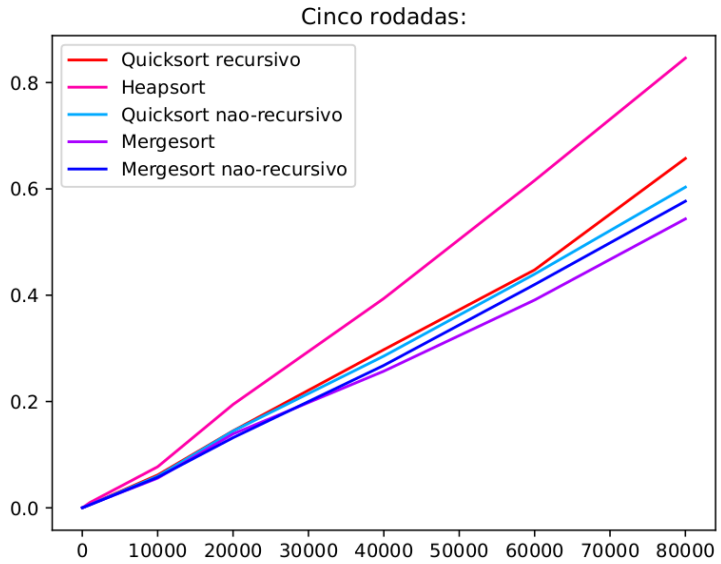
- *Uma rodada:*



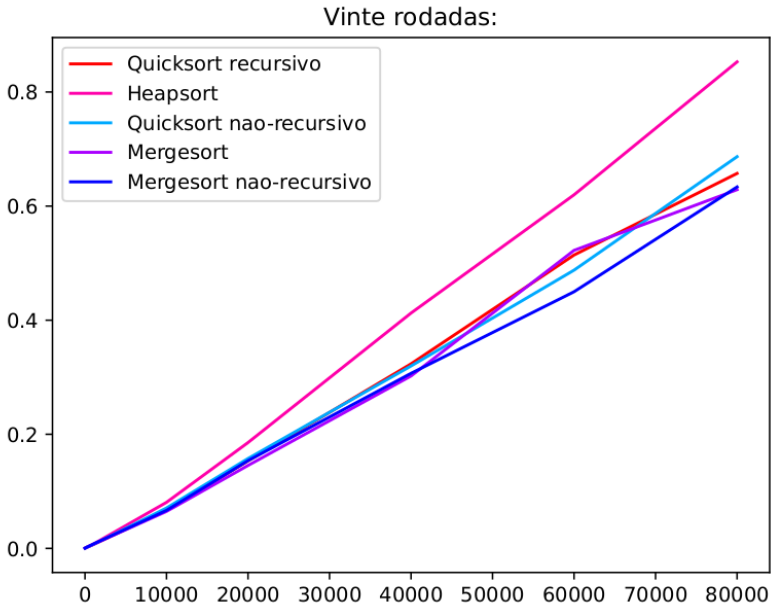
- *Duas rodadas:*



- *Cinco rodadas:*



- *Vinte rodadas (dez no caso de 10 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 mais do que o crescimento da quantidade de entradas das mesmas (veja a seção *Análise de Complexidade* para ver o porque). Como as medições foram feitas em 10, 100, 1000, 10000 e 100000 URLs, os gráficos acima podem parecer retas pela distância dos dois últimos pontos, mas observando os primeiros pontos é possível notar que o crescimento é um pouco mais que linear.

É interessante notar também que os mapas dos padrões de acesso à memória de cada algoritmo são bastante diferentes, sendo que o quicksort não recursivo apresentou melhor distância distância de pilha total (1825630 no caso de 1000 URLs e uma rodada), o que condiz com o fato de que o quicksort costuma ser mais rápido que os outros algoritmos, e implementações iterativas costumam ser melhores.

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.

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 *ordena*. É preciso passar para o programa os seguinte argumentos:

- *-i <arq>* para informar o caminho do arquivo de entrada do programa.
- *-o <arq>* para informar o caminho do arquivo de saída do programa.
- *-f <num>* para informar a quantidade de fitas a utilizar.
- *-n <num>* para informar a quantidade de entidades a permitir simultaneamente na memória primária.
- *-p <arq>* para informar o nome do arquivo de registro de desempenho.

Além de ser possível informar os seguintes argumentos opcionais:

- -h para obter ajuda sobre o uso do programa.
- *-l* para informar se é desejado registrar os acessos à memória. Caso essa opção não seja passada, será registrado somente o tempo de execução.
- *-d [1_merge, 1_heap, 2_quick, 2_merge]* para informar se é desejado que algum dos algoritmos dos desafios seja usado, e qual deles.
- *-c* para informar se é desejado registrar apenas acessos para comparações entre elementos. Caso essa opção não seja passado, serão registrados todos os acessos aos elementos.

É importante destacar também que o arquivo de entrada do programa deve seguir a seguinte especificação (o arquivo de saída também a seguirá):

- Cada linha do arquivo deve possuir um URL seguido de um espaço e o número de acessos a esse URL (logo em seguida deve haver quebra de linha).