Trabalho Prático 2
Ordenação em Memória Externa

Flávio Marcílio de Oliveira

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

1. INTRODUÇÃO

Devido a grande quantidade de dados gerados todos os dias, está cada vez mais difícil processar e extrair informações relevantes deste grande volume de dados. Em uma plataforma de Web Analytics, por exemplo, uma das tarefas é detectar quais os sites mais populares e ordená-los. Um arquivo contendo URLs e o número de visitas que cada uma recebe pode ter diversos GB de tamanho, excedendo facilmente a memória principal (RAM) de um computador pessoal. Portanto, para executar esta tarefa é necessário utilizar métodos de ordenação em memória externa.

Este documento apresenta o projeto de um algoritmo de ordenação dividido em duas etapas. Na primeira etapa, o algoritmo lê do arquivo de entrada um determinado número de entidades (URLs e números de visualizações) que serão ordenadas na memória principal. Este conjunto é ordenado utilizando o método de ordenação QuickSort e gravado na memória secundária. Este processo é repetido até que o arquivo de entrada tenha sido lido por completo e gerado n arquivos com os respectivos conjuntos de entidades ordenados. Na segunda etapa, os n arquivos gerados na primeira etapa são intercalados gerando o arquivo final com todas as entidades ordenadas. Para o processo de intercalação, na segunda etapa, uma entidade é lida de cada arquivo e colocada em um Heap e, dessa forma, garante-se que a maior entidade seja a primeira a ser retirada para ser gravada no arquivo final. Além disso, uma nova entidade é lida do mesmo arquivo do qual a entidade escrita no arquivo final veio e esse processo é repetido até que o Heap fique vazio e o arquivo tenha sido todo ordenado.

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

2. IMPLEMENTAÇÃO

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

2.1. Estrutura de Dados

Para este projeto foram desenvolvidos os TADs Item para representar as entidades a serem ordenadas, Heap para representar a Fila de Prioridade utilizada na segunda etapa do algoritmo, Ordenação para realizar a ordenação na primeira etapa e OrdenacaoExterna responsável pela segunda etapa. A seguir são apresentadas as características dos TADs com suas respectivas operações.

2.1.1. TAD - Item

Esse TAD foi desenvolvido por meio de uma Classe do C++, com três atributos: ***url_*** para as URLs, ***views_*** para quantidade de visualização da URL e ***fita_*** para identificar em qual fita esta entidade é gravada. Além dos atributos, O TAD possui os métodos: ***construtor*** com parâmetros - para a criação da entidade, ***compara*** - para realizar a comparação entre duas entidades, ***toString*** - para converter a entidade em uma string e ***Fita*** - que retorna o número da fita da entidade.

2.1.2. TAD - Heap

Heap é uma árvore binária completa e de prioridade. Uma árvore binária diz-se completa quando os seus níveis estão cheios, com possível exceção do último, o qual está preenchido da esquerda para a direita até um certo ponto. Uma árvore binária diz-se de prioridade quando o valor de cada nó tem maior ou igual prioridade que os seus filhos. Assim a raiz apresentará sempre o valor de prioridade máxima.

Este TAD foi desenvolvido utilizando uma Classe com os métodos: ***construtor*** com parâmetros que cria um Heap a partir de um arranjo, ***add*** para adicionar um novo elemento ao Heap, ***empty*** que verifica se o Heap está vazio e ***pop*** que retira o elemento de maior prioridade. Além desses métodos, foram criados os atributos ***id_*** para utilizar na análise de Localidade de Referência, ***HeapArrayPtr*** ponteiro para ponteiros que indica o Heap, ***MaxHeapSize*** para indicar o tamanho máximo do Heap e ***HeapSize*** para indicar o tamanho atual do Heap. Ainda para auxiliar na construção e manipulação do Heap foram criados os métodos ***FilterDown*** que percorre o Heap em direção às folhas refazendo o Heap , ***FilterUp*** que percorre o Heap em direção à raiz refazendo o Heap, ***Parent*** que retorna o índice do nó pai de um nó filho dado, ***RightChild*** que retorna o índice do filho à direita e ***LeftChild*** que retorna o índice do filho à esquerda.

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

2.1.3. TAD - Ordenacao

Esse TAD foi implementado como uma Classe com os seguintes métodos: ***construtor*** com parâmetros que chama o método de ordenação para o array passado. Como métodos auxiliares foram desenvolvidos ***Particao*** que divide a entrada em duas partições, ***Ordena*** que faz a ordenação de cada uma das partições criadas, ***QuickSort*** chama o método ***Ordena*** para o conjunto inteiro de dados e ***MedianaDeTres*** que encontra o índice do pivô utilizado na partição utilizando a estratégia da mediana-de-três para evitar o pior caso no processo de ordenação.

Para facilitar a definição do método de ordenação utilizado, foi criado um ***enum*** contendo os métodos *quicksort*, *mergesort* e *heapsort* para implementações futuras na escolha do método de ordenação.

A implementação desse TAD foi desenvolvida utilizando o Padrão de Projeto Template.

2.1.4. TAD - OrdenacaoExterna

Esse TAD é o responsável por realizar todo o processo de ordenação do arquivo de entrada e foi implementado com os métodos: ***construtor*** com parâmetro responsável por gerar as rodadas e chamar o método ***Intercala*** para gerar o arquivo ordenado final, ***Intercala*** responsável por fazer a intercalação das entidades e criar o arquivo final, ***writeFile*** responsável por verificar a condição dos arquivos para a escrita, ***readFile*** responsável por verificar a condição dos arquivos de leitura.

2.2. Organização do Código

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

2.3. Funcionamento do Programa

Para uma melhor compreensão do projeto, esta seção apresenta a descrição do funcionamento de cada função dos TADs.

2.3.1. Operações do TAD Item - item.cpp

Construtor - recebe como parâmetro uma string representando a URL, um inteiro representando o número de visualizações e um inteiro que representa o número do arquivo que será gravado. Com os parâmetros constrói um objeto Item.

Compara - método utilizado para comparar dois objetos do tipo Item. A comparação é feita inicialmente pelo número de views e, caso seja igual, compara as URLs utilizando o método ***compare*** da biblioteca string do C++.

toString - método utilizado para converter o objeto Item em uma string e, para isso, retorna a concatenação dos atributos ***url_*** e ***views_*** com um espaço entre eles.

Fita - retorna o atributo ***fita_***.

2.3.2. Operações do TAD Heap - heap.cpp

Construtor - recebe como parâmetros um array de ponteiros, a quantidade de objetos neste array e um id e faz a atribuição dos atributos ***id_***, ***MaxHeapSize***, ***HeapSize*** e ***HeapArrayPtr***. Depois calcula o índice do último nó da árvore que não é folha e chama o método ***FilterDown*** para construir o Heap em um laço que vai subindo na árvore construindo o Heap até chegar no nó raiz.

add - recebe um item e insere na última posição disponível do Heap e depois chama o método ***FilterUp*** para verificar e reconstruir o Heap com as suas propriedades.

empty - verifica se o Heap está vazio retornando um booleano com o resultado.

pop - retira o elemento da raiz, coloca o último elemento na posição da raiz e chama o método ***FilterDown*** passando o índice zero, de modo a refazer o Heap.

FilterDown - verifica se os filhos de um nó pai (passado como parâmetro) são maiores, e caso sejam, compara qual filho é maior e faz a troca com o nó pai. Este método percorre até os nós folhas, descendo na árvore, buscando o maior filho de modo a efetuar a troca com o maior filho.

FilterUp - verifica se o nó pai de um nó filho (passado como parâmetro) é menor, e caso seja, é feita a troca. Esse processo é repetido até que se encontre um nó pai maior de modo que não precise fazer a troca e o Heap está refeito com suas propriedades mantidas.

Parent - método que retorna o índice do nó pai de um determinado nó.

RightChild - método que retorna o índice do filho à direita de um determinado nó.

LeftChild - método que retorna o índice do filho à esquerda de um determinado nó.

2.3.3. Operações do TAD Ordenacao - ordenacao.cpp

Construtor - recebe como parâmetro um array de ponteiros, um inteiro com a quantidade de elementos deste array e um dos três métodos de ordenação (*quicksort, mergesort, heapsor*t) disponíveis para escolha (obs.: até este momento só o *quicksort* está implementado e é o que será executado). A partir dos parâmetros passados, o método de ordenação é escolhido passando o array e a quantidade de elementos.

Particao - faz uma chamada para o método ***MedianaDeTres*** para determinar o índice do pivô. Com o pivô determinado, são criadas duas partições: a partição esquerda com os elementos maiores que o pivô e a partição direita com os elementos menores que o pivô.

Ordena - método recursivo que inicialmente cria duas partições e faz chamadas recursivas para as duas partições criadas. O método consiste em particionar o conjunto em subconjuntos ordenados, repetindo esse processo sucessivamente até que reste apenas um único elemento. Ao fim deste processo o conjunto estará totalmente ordenado.

QuickSort - método que chama o método ***Ordena*** para o conjunto inteiro de dados.

MedianaDeTres - método utilizado para encontrar o índice do pivô. A estratégia consiste em utilizar três elementos do conjunto e calcular a mediana entre estes três elementos. Usualmente são utilizados o primeiro, o elemento do meio e o último elemento do conjunto para encontrar a mediana, o que será adotado neste trabalho. A opção por esta estratégia na escolha do pivô se deve a diminuir a chance de cair no pior caso em que o método quicksort tenha uma complexidade $\mathbf{O(n^2)}$ comparações.

2.3.4. Operações do TAD OrdenacaoExterna - ordenacao_externa.cpp

Construtor - recebe como parâmetros o nome do arquivo de entrada de dados, o nome do arquivo de saída e o número de entidades que serão lidas para a memória interna em cada rodada de execução. Nesse método, em cada rodada são lidas a quantidade determinada de entidades e colocada em um arranjo que em seguida é ordenado utilizando o método *quicksort* e após gravado na memória externa em um arquivo. Esse processo é repetido até que o arquivo de entrada tenha sido totalmente lido, gerando-se vários arquivos com blocos de dados ordenados. Após a geração desses diversos arquivos, o método ***Intercala*** é chamado passando a quantidade de arquivos gerados e o nome do arquivo de saída.

Intercala - recebe a quantidade de arquivos gerados com partes de dados ordenados e o nome do arquivo de saída onde será gravado o resultado com todas as entidades ordenadas. Para a execução deste processo de intercalação, uma entidade de todos os arquivos gerados na etapa anterior é lido para um array e criado um heap. Depois dessa fase inicial de criação do heap é feito um laço de execução consistindo em retirar um elemento do heap e escrever no arquivo de saída final e logo em seguida uma nova entidade é lida do arquivo da mesma entidade retirada do heap. Esse processo continua até que todas as entidades de todos os arquivos tenham sido lidas e o heap fique vazio. No final, tem-se o arquivo de saída com todas as entidades ordenadas.

writeFile - método que verifica se o arquivo onde os dados são gravados está aberto e com condições de receber novos dados.

readFile - método que verifica se o arquivo está com condições para ser lido.

3. ANÁLISE DE COMPLEXIDADE

3.1. Complexidade de Tempo

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

3.1.1. TAD - Item

Construtor - método que realiza apenas três atribuições, portanto, **O(1)**.

Compara - método que faz uma comparação inteira no melhor caso quando o número de visualizações é diferente, portanto, **Ω(1)** e no pior caso, o número de visualizações é igual e é necessário comparar as URLs e, neste caso, utiliza a função ***compare*** da biblioteca string da Linguagem C++ que, segundo a documentação, é **O(n)** onde **n** é o tamanho da string.

toString - método que concatena uma string, um espaço e um inteiro convertido para string. Operações simples com complexidade **O(1)**.

Fita - apenas retorna um valor, sendo **O(1)**.

3.1.2. TAD - Heap

Construtor - faz atribuições e chama o método ***FilterDown*** *(n-2)/2* vezes, logo, **O(n log n)**;

add - faz uma chamada ao método ***FilterUp***, portanto, **O(log n)**;

empty - faz apenas uma comparação, portanto, **O(1)**;

pop - faz uma chamada do método ***FilterDown***, portanto, **O(log n)**;

FilterDown - no pior caso percorre todo um galho da árvore binária, ou seja, executa *log n* operações, portanto, **O(log n)**;

FilterUp - no pior caso percorre todo um galho da árvore binária, portanto, **O(log n)** assim como o método ***FilterDown***;

Parent - método que calcula um índice, portanto, **O(1)**;

RightChild - método que calcula um índice, portanto, **O(1)**;

LeftChild - método que calcula um índice, portanto, **O(1)**;

3.1.3. TAD - Ordenacao

Construtor - método que faz uma comparação e chama o método ***QuickSor***t, portanto, **O(n log n)**;

MedianaDeTres - executa um número fixo de comparações, independente da entrada, portanto, **O(1)**;

Particao - faz uma chamada ao método ***MedianaDeTres*** e faz **n** comparações, portanto, **O(n)**;

Ordena - método recursivo com função de complexidade dada por $C(n) = 2C(n/2) + n$ pois faz duas chamadas recursivas para as duas partições criadas pelo método ***Particao***. O método, neste caso, tem uma complexidade **O(n log n)**, uma vez que foi utilizado a estratégia da escolha do pivô utilizando a mediana de três para evitar o pior caso, que teria uma complexidade $\mathbf{O(n^2)}$;

QuickSort - método que chama o método ***Ordena*** passando o array todo, portanto, complexidade do método ***Ordena***, ou seja, **O(n log n)**.

3.1.4. TAD - OrdenacaoExterna

Construtor - Para a geração de **m** arquivos contendo cada um **n** entidades ordenadas, este método irá fazer **m** chamadas ao método construtor do TAD ***Ordenacao***, gerando um custo total de **O(mn log n)**;

Intercala - faz uma chamada inicial do construtor do Heap, com custo **O(n log n)**, e outras **n** chamadas dos métodos ***pop*** e ***add*** do Heap, ambos com custo **O(log n)**. Neste método, foi implementado um laço para verificar de qual arquivo uma nova entidade deve ser lida quando o arquivo de origem da entidade retirada do Heap não possuir mais elementos. Este processo tem um custo global **O(m)** considerando **m** arquivos. Portanto, o método ***Intercala*** tem um custo global, **O(n log n) + n(O(log n) + O(log n)) + m = O(n log n + m)**;

writeFile - método **O(1)** pois só faz a verificação da condição do arquivo de escrita;

readFile - método **O(1)** pois só faz a verificação da condição do arquivo de leitura;

3.2. Complexidade de espaço

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

3.2.1. TAD - Item

Construtor - armazena uma string e dois inteiros. Como a string pode ter qualquer tamanho dependendo da URL passada, a complexidade é **O(n)**;

Compara - **O(1)** pois não necessita de nenhuma memória extra para a execução.

toString - **O(1)** pois não armazena nenhum dado.

Fita - **O(1)** pois só retorna um valor.

3.2.2. TAD - Heap

Construtor - **O(1)** pois a memória utilizada é independente do tamanho da entrada;

add - utiliza memória para armazenar um novo elemento. Considerando elementos de tamanho **n**, a complexidade de espaço deste método é **O(n)**;

empty - não utiliza memória extra, portanto, **O(1)**;

pop - utiliza memória extra temporária para retornar o elemento retirado, portanto, **O(n)** onde **n** é o tamanho do elemento.

FilterDown - **O(n)** pois utiliza memória extra para auxiliar na troca de posição dos elementos;

FilterUp - **O(n)** pois utiliza memória extra para auxiliar na troca de posição dos elementos;

Parent - **O(1)** pois apenas retorna um valor sem utilização de memória adicional;

RightChild - **O(1)** pois retorna um valor sem necessidade de memória adicional;

LeftChild - **O(1)** pois retorna um valor sem necessidade de memória adicional.

3.2.3. TAD - Ordenacao

Construtor - não utiliza memória adicional, portanto, **O(1)**;

Particao - **O(1)** pois só utiliza manipulação de ponteiros para um tipo de elemento;

Ordena - **O(1)** pois não utiliza memória adicional;

QuickSort - **O(1)** pois não utiliza memória adicional;

3.2.4. OrdenacaoExterna

Construtor - sendo **n** o número de entidades lidas em cada rodada e **m** o tamanho de cada entidade, a complexidade de espaço deste método será **O(nm)**;

Intercala - considerando **n** fitas e que cada entidade tem um tamanho **m**, a complexidade deste método será **O(nm)**;

writeFile - **O(1)** pois não utiliza memória adicional;

readFile - **O(1)** pois não utiliza memória adicional.

4. ESTRATÉGIAS DE ROBUSTEZ

Para garantir a tolerância às falhas e visto que os TADs projetados são essenciais para a execução do programa, optou-se por desenvolver uma estratégia de lançar os erros ocorridos

na saída padrão de erro e encerramento do programa com código 1, evitando prosseguir a execução do código com entradas inválidas.

5. ANÁLISE EXPERIMENTAL

A análise experimental foi desenvolvida para o *Ordenador* buscando avaliar tanto o desempenho em questão de tempo de execução quanto para a Localidade de Referência. Os arquivos de testes foram gerados pelo programa ***geracarga***, disponibilizado pelo professor no Moodle.

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

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

RAM: 8,00GB

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

WSL2: Ubuntu-20.04

5.1. Análise de Desempenho

Para a análise de desempenho, foram gerados 10 arquivos com quantidades diferentes de entidades aleatórias e para cada arquivo foram realizadas 10 execuções, e com isso, calculado o tempo de execução como a média destas execuções, conforme a Tabela 1. Para efeitos de comparação e análise, considerou-se constante o número de entidades lidas (50) para todos os arquivos.

Tabela 1 - Tempo de execução experimentais

Tamanho da Entrada (entidades)	Execuções										Tempo médio
	1	2	3	4	5	6	7	8	9	10	
100	0.1589375	0.1246257	0.1392181	0.1046896	0.1104250	0.1303086	0.1001695	0.1024341	0.1100413	0.1272345	**0.1208084**
200	0.2942714	0.3660902	0.2095675	0.2161295	0.2541709	0.2166814	0.1880363	0.2123005	0.1765258	0.1978568	**0.2331630**
300	0.2745902	0.3116490	0.3270865	0.2372015	0.2794414	0.3157948	0.3148428	0.4864458	0.3099003	0.2872516	**0.3144204**
400	0.3752497	0.3909870	0.3856237	0.3758930	0.4086388	0.3811463	0.4007560	0.4370930	0.5421053	0.3988689	**0.4096362**
500	0.4239715	0.4735443	0.5991830	0.4768771	0.5117775	0.4391881	0.4775513	0.5164929	0.5624388	0.4447481	**0.4925773**
600	0.5847212	0.5372097	0.5330741	0.6023573	0.5865840	0.5490629	0.5706304	0.5764452	0.6011422	0.5391894	**0.5680416**
700	0.6299626	0.6259198	0.6285052	0.7597848	0.6701130	0.6539455	0.6559168	0.7411787	0.6489099	0.6928731	**0.6707109**
800	0.7203572	0.7218494	0.8009749	0.7574851	0.8106576	1.0781174	0.7141045	0.7044830	0.7927484	0.7800925	**0.7880870**
900	0.7364120	0.7674158	0.7931882	0.7803629	0.7951037	0.7722784	0.6744077	0.8357499	0.7522849	0.7148332	**0.7622037**
1000	0.8613561	0.9159093	0.9359202	0.8653652	0.9168489	0.8338547	0.8831284	0.9296039	0.8793675	0.9120156	**0.8933370**

Na Figura 1 é apresentado o gráfico do tempo de execução em função do número de entidades a serem ordenadas. Neste gráfico também é apresentado o comportamento assintótico esperado para o Ordenador, que neste caso é **O(n log n)** pois a quantidade de entidades lidas é mantida constante variando o número de arquivos intermediários (fitas) utilizados.

A geração do gráfico com o comportamento teórico foi criado utilizando uma mudança de escala para possibilitar a comparação dos dois comportamentos.

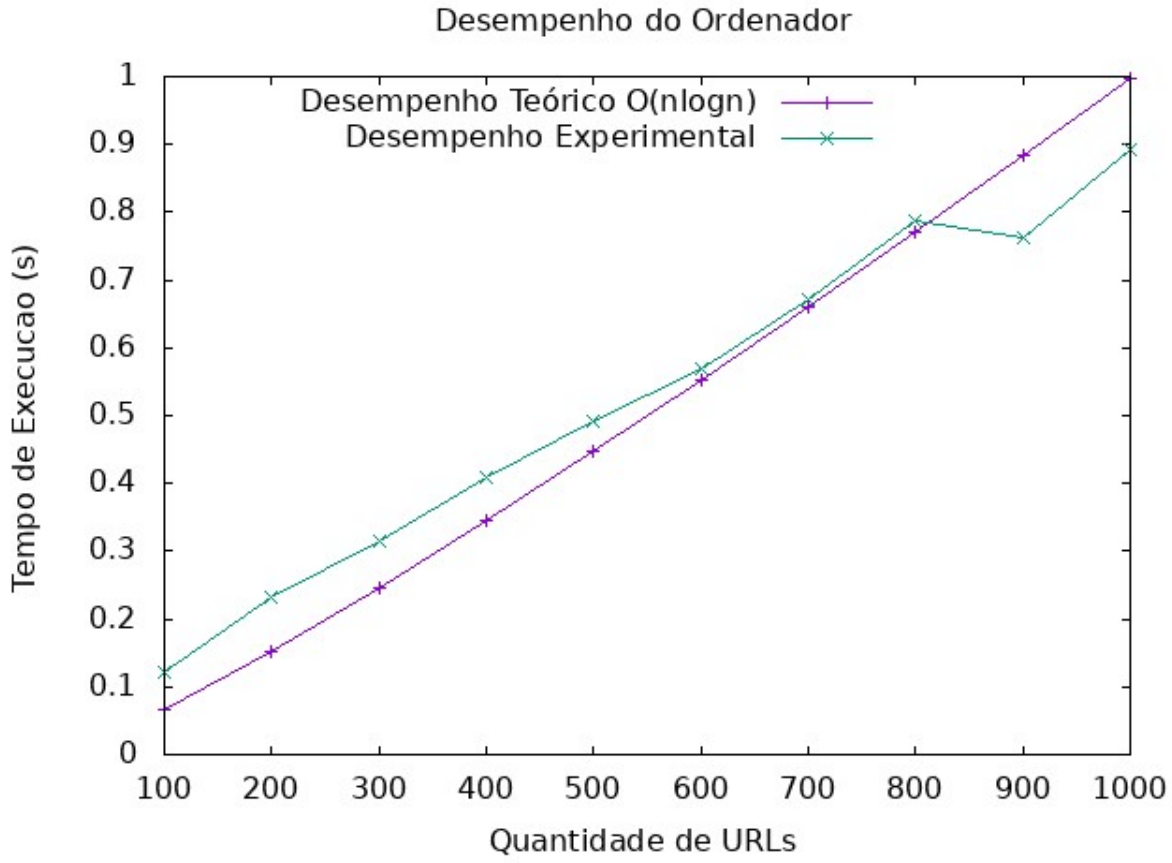

Figura 1 - Tempo de execução em função do tamanho

Pela análise do gráfico apresentado na Figura 1, verifica-se que o comportamento real do algoritmo desenvolvido segue o comportamento previsto.

5.2. Análise de Localidade de Referência

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

5.2.1. Mapa de Acesso à Memória

Para a análise de Localidade de Referência foi utilizado um arquivo com 100 entidades sendo lido 50 entidades a cada rodada, portanto, gerando duas rodadas. Além disso, como o QuickSort é chamado a cada rodada definiu-se os IDs 0 e 1 para cada processo de ordenação e o ID 30 para identificar o Heap utilizado no processo de intercalação.

A Figura 2 apresenta o mapa de acesso para a primeira ordenação utilizando o quicksort.

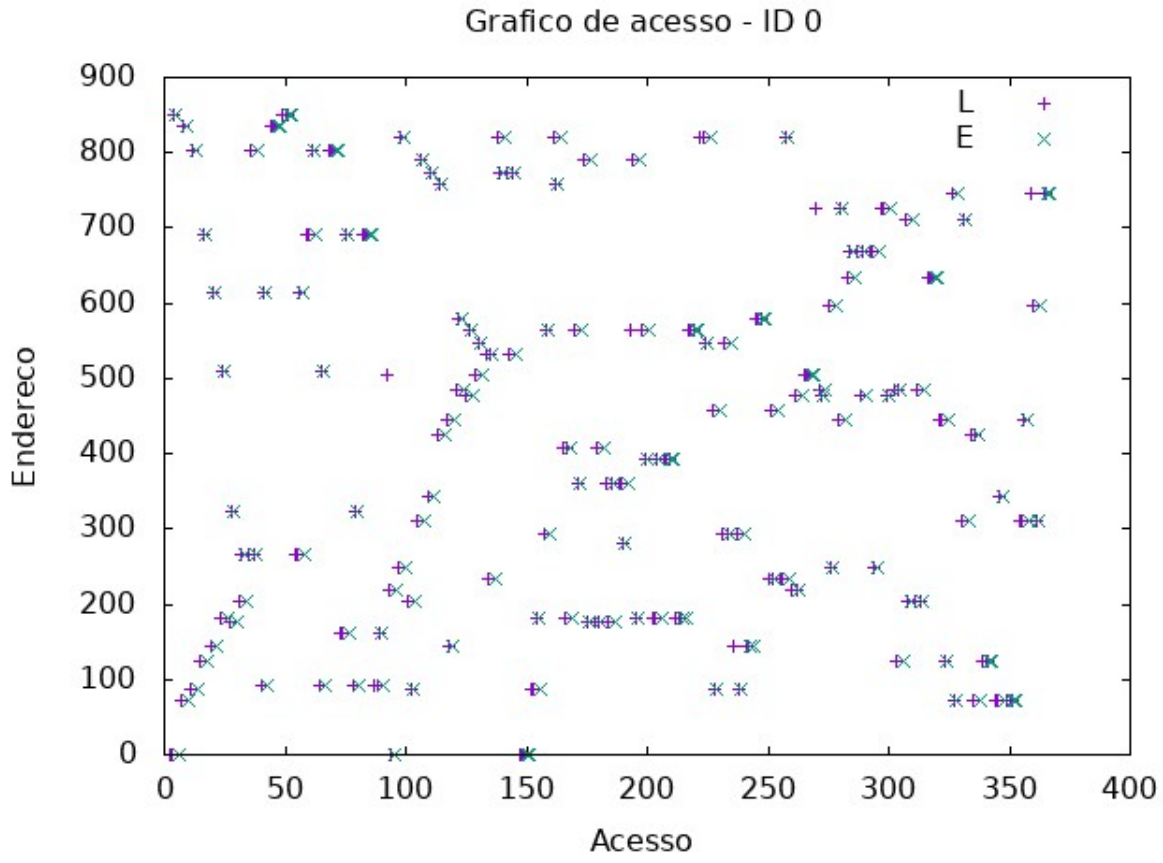

Figura 2 - Mapa de Acesso à memória para as 50 primeiras entidades

Pela análise da Figura 2 é possível concluir que, para as 50 primeiras entidades lidas, o processo de ordenação exige uma relativa movimentação da memória, mostrando que esta primeira parte do arquivo tem uma distribuição aleatória das entidades.

A Figura 3 mostra o mapa de acesso para o processo de ordenação da segunda parte do arquivo.

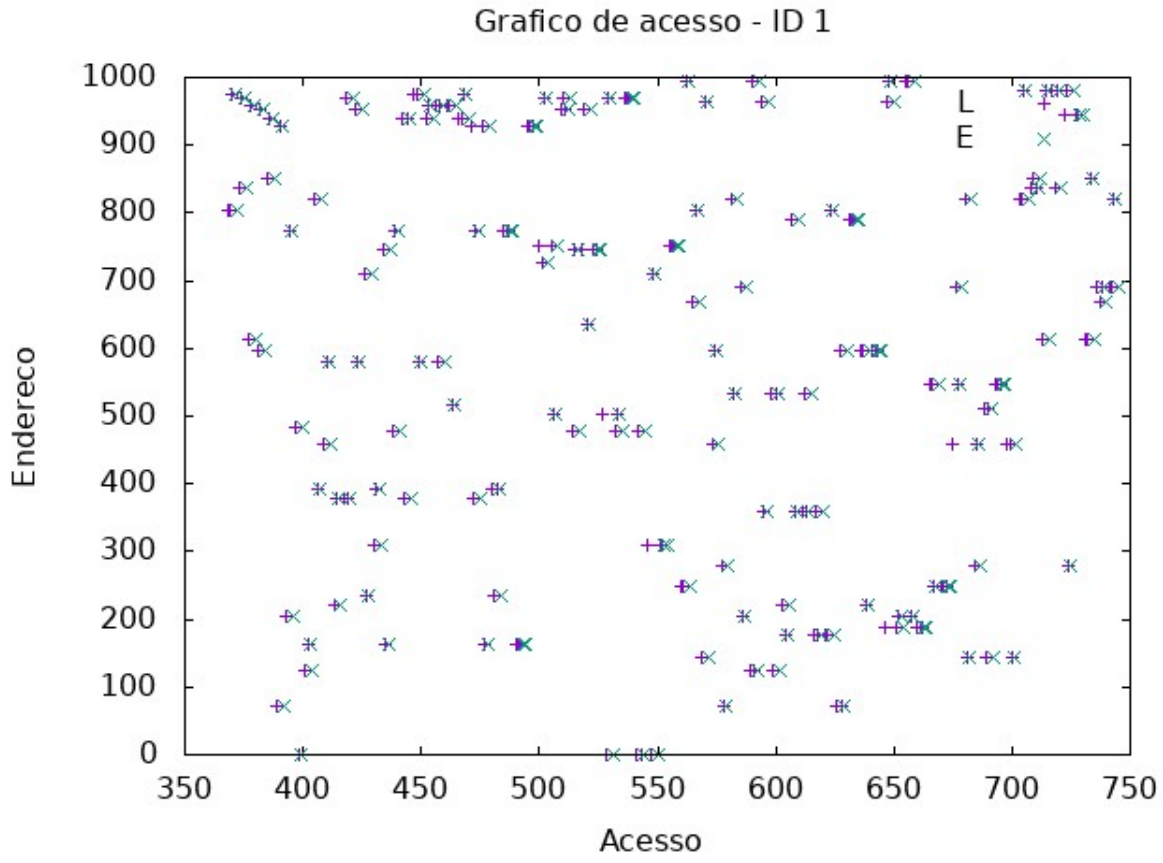

Figura 3 - Mapa de Acesso à memória para as 50 últimas entidades

Pela análise da Figura 3 também é possível concluir que esta parte do arquivo também apresenta uma distribuição aleatória das entidades sendo necessário uma grande movimentação na memória para a realização da ordenação.

A Figura 4 apresenta o mapa de acesso à memória para o Heap utilizado no processo de intercalação, onde as entidades são lidas dos arquivos gerados na etapa anterior pelo quicksort. Pela estratégia utilizada na implementação neste trabalho, inicialmente são lidas as entidades dos arquivos e armazenadas em um arranjo e, só então é chamado o método para a criação do Heap passando este arranjo. Assim, neste primeiro momento, a movimentação no Heap ocorre somente para garantir as propriedades. Nas etapas seguintes tem-se a retirada do elemento da raiz e a inserção de novo elemento e, com isso, o método ***add*** só é chamado nesta etapa.

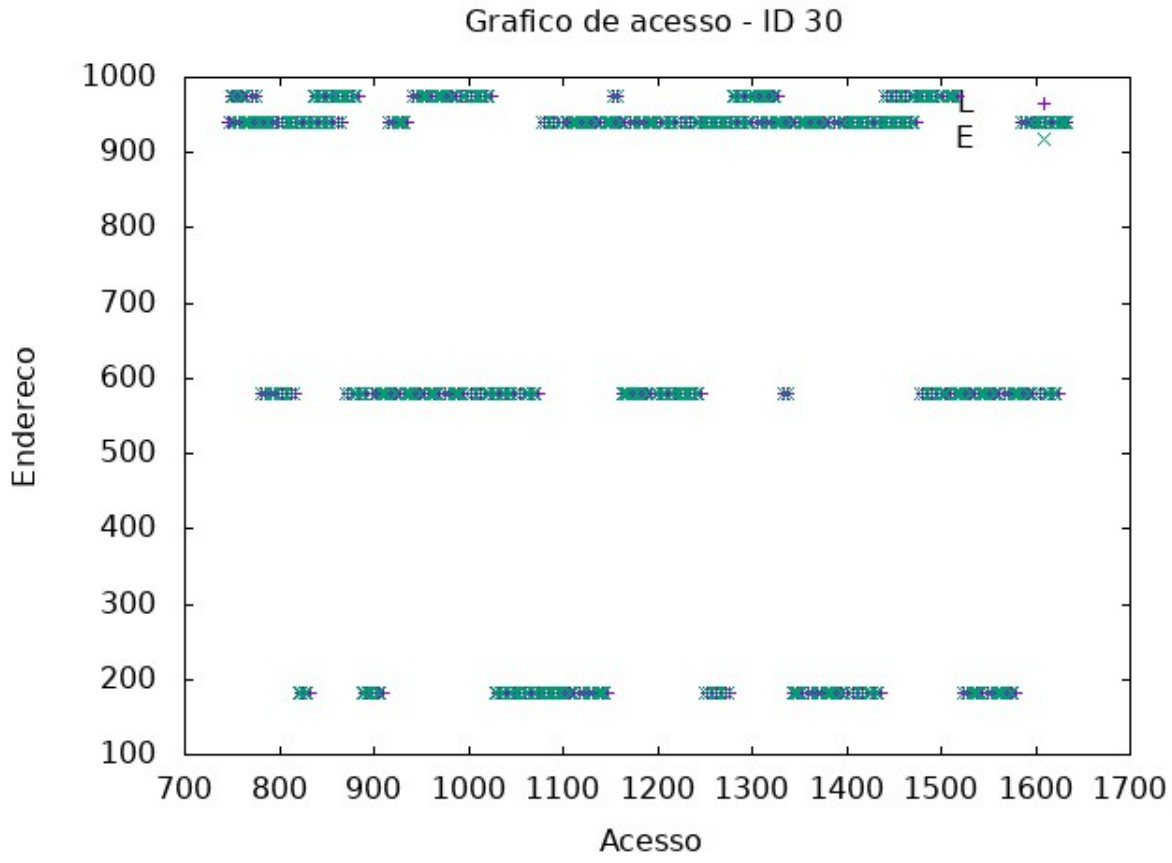

Figura 4 - Mapa de Acesso à memória para o Heap

Pela análise da Figura 4 observa-se que o Heap utiliza essencialmente quatro posições na memória e a leitura e escrita de novas entidades são feitas nestas posições comprovando o princípio da Localidade de Referência. Isso ocorre devido a implementação utilizada no projeto, onde apenas uma entidade é lida de cada arquivo gerado.

5.2.2. Distância de Pilha

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

Para realizar a análise da distância de pilha, a execução foi dividida em duas fases: Fase 0 - geração das rodadas, utilizando o quicksort e; Fase 1 - intercalação das fitas geradas na Fase 0.

A Figura 5 mostra a distância de pilha para o primeiro processo de ordenação do quicksort.

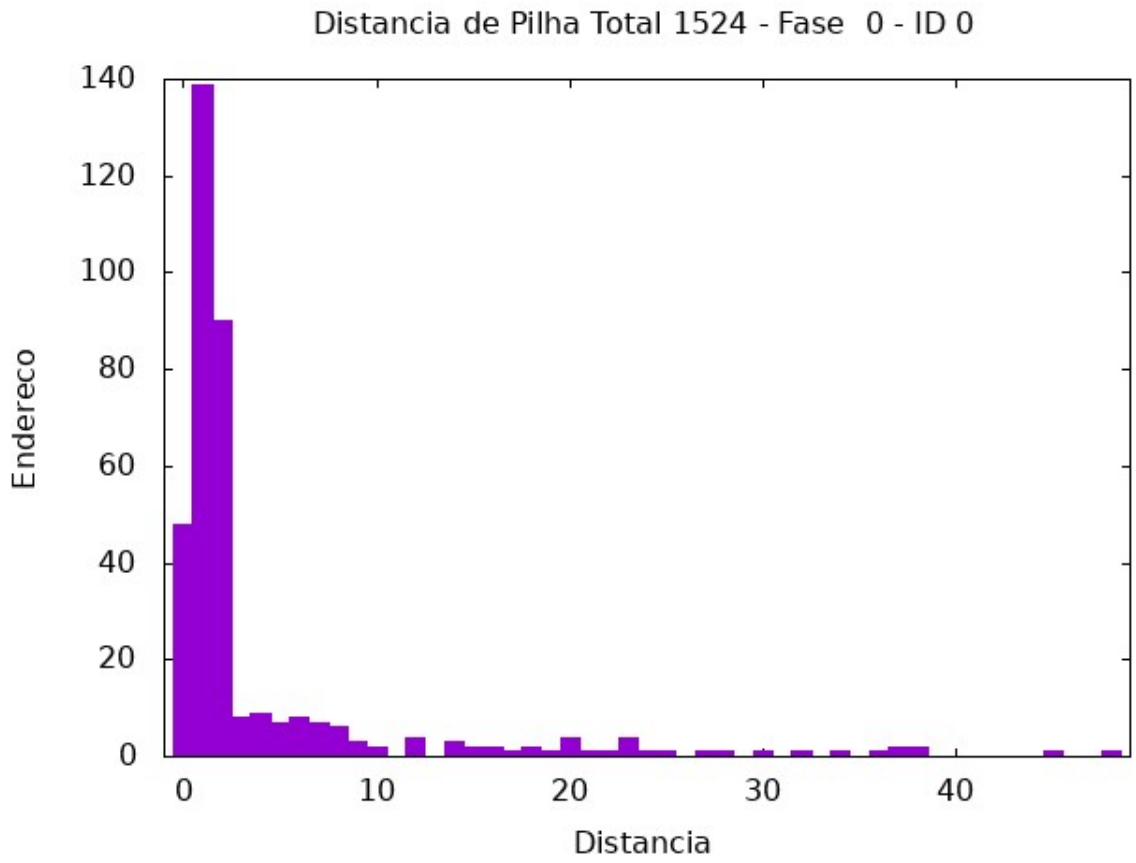

Figura 5 - Distância de pilha para a primeira ordenação com quicksort

Pela análise da Figura 5, observa-se que grande parte dos endereços de memória tiveram uma distância de pilha pequena enquanto poucos tiveram uma distância de pilha maior, indicando que o tempo entre dois acessos sucessivos de um endereço específico é muito pequeno para a maioria dos elementos ordenados no quicksort. Esse comportamento indica o uso eficiente da memória contribuindo para o bom desempenho em tempo de execução do algoritmo de ordenação.

A Figura 6 apresenta a distância de pilha para a ordenação do segundo conjunto de dados.

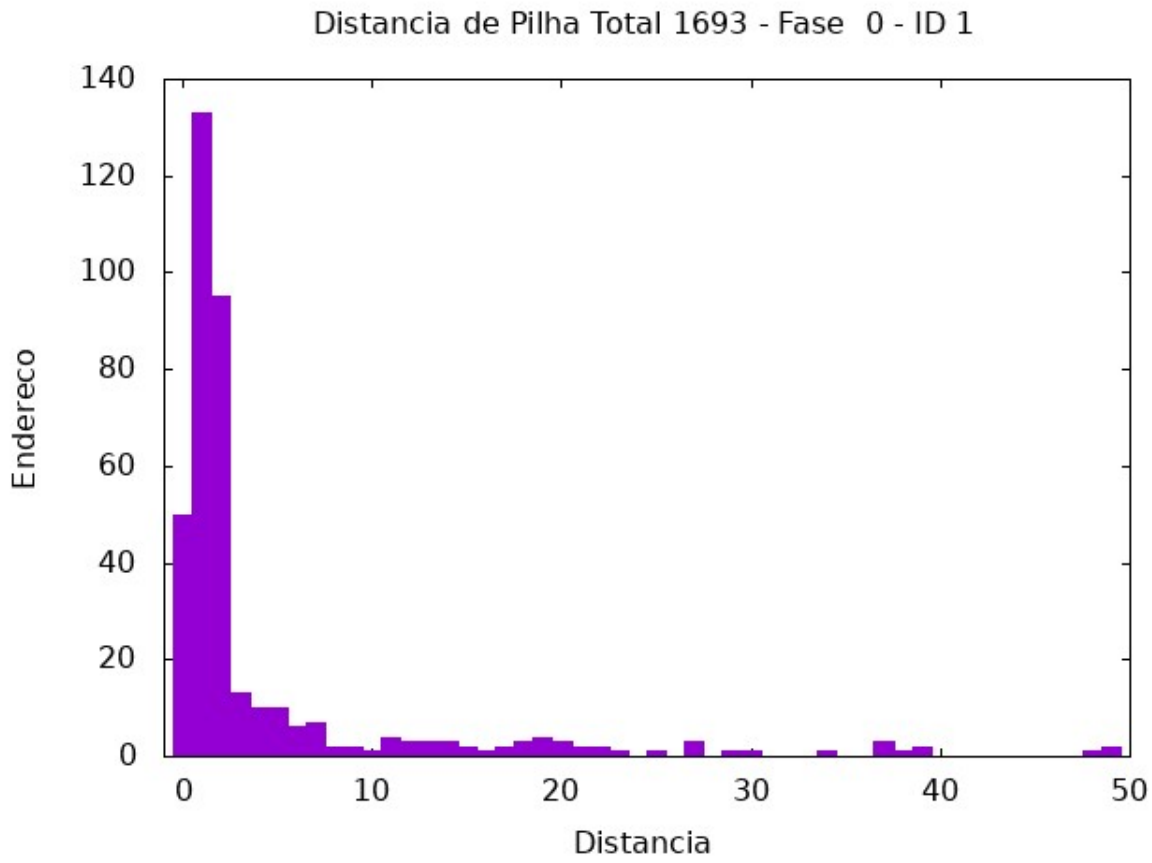

Figura 6 - Distância de pilha para a segunda ordenação do quicksort

O mesmo comportamento observado na primeira ordenação (Figura 5) é observado neste caso mostrado na Figura 6.

A Figura 7 mostra a distância de pilha para o Heap utilizado para intercalar os dois arquivos gerados na Fase 0 e gerar o arquivo final.

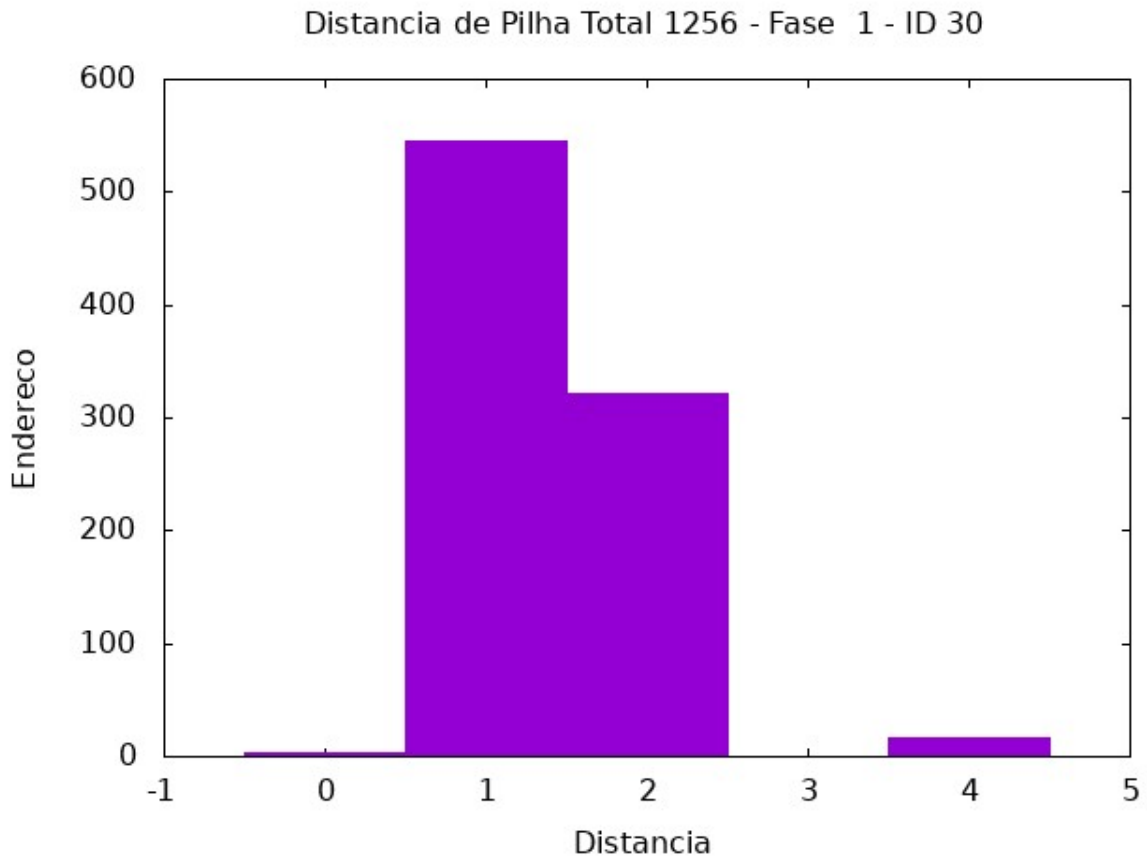

Figura 7 - Distância de pilha para o Heap

A Figura 7 reforça o que foi mostrado na Figura 4, onde observou-se a utilização de apenas quatro endereços de memória distintos.

6. Conclusão

Este trabalho apresentou a implementação e análise de um programa de ordenação em memória externa, fazendo a utilização do algoritmo Quicksort para a ordenação de partes menores do arquivo de entrada e, na segunda etapa, a utilização do Heap para fazer a intercalação dos arquivos gerados e criar o arquivo final com todos os dados ordenados.

O programa foi analisado quanto a complexidade de tempo e de espaço e testes de desempenho e de localidade de referência foram realizados para avaliar o comportamento global dos códigos desenvolvidos.

Para desenvolvimentos futuros e implementações de melhorias, seria interessante implementar outros métodos para a ordenação na primeira fase (geração das rodadas), como por exemplo, o mergesort e o heapsort e fazer uma comparação entre os três algoritmos. Outra melhoria que pode ser feita é implementar um algoritmo simples de ordenação (seleção ou inserção) para ganhar desempenho na ordenação de partições pequenas e também,

implementar etapas de intercalação intermediária com definição do tamanho máximo do Heap (método intercalação balanceada de vários caminhos[4]).

Bibliografia

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

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

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

[4] Ziviani, Nivio. Projeto de Algoritmos com implementações em Pascal e C. São Paulo: Pioneira, 1999

Apêndice A

Instruções de compilação e execução

A.1 Compilação

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

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

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

A.2 Execução

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

Dentro da raiz do projeto: **./bin/main** <*arquivo de entrada*> <*arquivo de saída*> **<**N**>**

onde: <*arquivo de entrada*> é o nome do arquivo que será lido com os dados a serem ordenados;

<*arquivo de saída*> é o nome do arquivo onde os dados ordenados serão gravados;

<N> é o número de entidades que será lido para a memória principal para a ordenação com o quicksort;

Registro de acesso

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