Documentação TP2 – Contador de Palavras

Jorge Augusto de Lima e Silva

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte – MG – Brasil

Professores: Wagner Meira, Giselle Pappa – Matéria: Estruturas de Dados (DCC205) – Semestre: 2022-1

E-mail: jorgesilva@dcc.ufmg.br

1- Introdução:

O problema proposto consistia de duas partes, a primeira é realizar a leitura de um arquivo de texto e contar o número de vezes que cada palavra é repetida nele, e a segunda parte, imprimir essa palavras de forma ordenada segundo uma redefinição da ordem lexical.

Para isso, foi necessário desenvolver uma forma de armazenar a nova ordem lexical e redefinir o operador de comparação entre duas strings. Em seguida foi preciso criar uma forma de armazenar as palavras. Por fim, foi necessário implementar a função de ordenação QuickSort, seguindo os requisitos do trabalho.

Também foram precisos criar formas de otimização do QuickSort, levando em consideração que as chamadas recursivas feitas por esse tem um custo.

2- Método:

a- Estruturas de Dados:

Foram utilizadas duas estruturas de dados principalmente: uma árvore e uma lista estática. Além disso foi criado também uma struct chamada wordCounter, responsável por armazenar uma palavra e a quantidade de vezes que essa está presente no texto. Foi criada a classe lexOrder, ficando esta responsável por armazenar a nova ordem lexicográfica e definir as operações que são dependentes desta, como, por exemplo, o operador maior que “>” de duas strings. Como estrutura auxiliar foi implementado o Node, sendo esta a unidade básica da árvore e, cada Node também pode ser interpretado como uma árvore em si.

Enquanto a árvore é utilizada como sendo a estrutura que recebe as palavras, a lista pode ser observada como a estrutura que ordena as palavras. O motivo para se escolher o uso de duas estruturas ao invés de uma será mais bem detalhado na seção seguinte (Análise de Complexidade).

b- Função Main:

A função main tem um funcionamento bem simples. Primeiramente ela lê os parâmetros de entrada recebidos e os processa. Estes estarão detalhados no Apêndice A. Conhecendo a estrutura do arquivo de entrada, o qual começa com a string #ORDEM, é sucedido pela nova ordem lexicográfica, depois pela string #TEXTO, e por fim pelo texto, foi possível definir uma sequência de operações simples.

Após a leitura dos parâmetros, é feito a leitura da ordem lexical, a qual é armazenada e na classe lexOrder. Em seguida, é feita a leitura do texto, no qual cada palavra é capturada, passa por um tratamento, de forma que todas as letras da palavra sejam transformadas em minúsculas, caracteres especiais sejam removidos e, caso o último caractere da palavra seja um hífen, esta é concatenada com a seguinte, assumindo assim que houve um erro de digitação e que as duas palavras foram separadas indevidamente.

Tendo agora as palavras em uma forma padronizada, é feito um caminhamento através da árvore binária de busca implementada. O operador usado para comparar as strings ao inseri-las na árvore, é o operador maior que “>” padrão da classe string, de modo que a árvore se encontre ordenada em ordem alfabética. Durantes esse caminhamento, são observados os possíveis locais que a palavra pode estar, não sendo consultados os lugares que ela não poderia estar em situação alguma, portanto, caso a palavra seja encontrada, o contador do nó que está inserido na árvore é incrementado, e, caso a palavra não seja encontrada, é adicionada uma nova folha que ficará responsável por armazenar a palavra e seu contador.

Assim que todas as palavras forem devidamente inseridas, a árvore é convertida em uma lista estática, seguindo o método in-ordem de caminhamento, facilitando assim a ordenação sobre esta.

Convertida em uma lista estática, esta por fim é ordenada e impressa no arquivo, concluindo assim a execução do programa.

3- Análise de Complexidade:

Durante essa seção serão feitas as análises de complexidade das funções principais presentes em cada uma das classes. Ao final dessa seção também será discutida a escolha de uma árvore como sendo a estrutura que recebe as inserções e, posteriormente a sua conversão em uma lista, sobre a escolha de se utilizar uma lista desde o início do processo.

a- Ordem Lexical:

As funções da classe ordem lexical são quase todas da ordem de O(1), sendo, em sua maioria, ou uma conversão de valores, ou uma consulta de valores. A única função que não tem custo constante é a função biggerThan(string,string). Essa função percorre os caracteres de cada uma das duas strings até encontrar um diferente ou até que uma das strings termine. Desta forma, recebendo uma string de tamanho m e outra de tamanho n, o custo dessa função é O(min(m,n)).

Outra característica interessante de se observar da classe lexOrder é que a consulta aos valores redefinidos da ordem lexicográfica tem custo O(1). Isso foi feito sabendo que exatamente as 26 letras do alfabeto teriam seus valores redefinidos. Portanto, foi criado um vetor de inteiros de 26 posições, no qual o n-ésimo elemento do vetor representa o valor redefinido da n-ésima letra do alfabeto, ou seja, a posição 0 do vetor contém o valor redefinido da letra a, a posição 1, da letra b, a 2, da letra c, e assim por diante até que se chegue na letra z. A partir dessa estrutura, para descobrir o novo valor de alguma letra do alfabeto, realizar uma consulta neste vetor, e, como o vetor é estático e se tem o índice, a própria letra, é possível ser realizado o acesso direto à informação desejada.

b- Node:

Ambos os métodos construtores do nó tem custo O(1), apenas definindo a palavra que será armazenada neste e inicializando as referências a seus filhos com valores nulos ou valores definidos pelo usuário.

Diferentemente do método construtor, o destrutor faz uma chamada recursiva para o destrutor de seus filhos, de modo sua complexidade dependa da quantidade de filhos que este nó tem no total. Tendo função de custo definida, aproximadamente como $T(n) = 2T(n/2)+1$, temos que a complexidade assintótica de se destruir um nó é da ordem de $O(n)$, pelo Teorema Mestre.

c- Árvore:

Assim como o construtor da classe nó, o construtor da classe Árvore é também é $O(1)$, sendo este responsável por inicializar a quantidade de elementos da árvore como sendo 0 e a raiz da árvore como sendo um ponteiro nulo.

O método de inserção, por sua vez, consiste em: (1) verificar se o valor do nó atual é igual ao valor que está sendo inserido, se for esse o caso o contador é incrementado e o método é interrompido, (2) caso o valor que está sendo inserido seja menor que o valor do nó atual, as comparações são refeitas, mas agora entre o valor que está pra ser inserido e o filho a esquerda do nó que foi comparado anteriormente, (3) faz-se a mesma coisa que em (2), porém observa se é maior e a nova comparação é feita com o nó a direita. Desta forma, fica evidente que, no melhor caso, a inserção tem custo $O(1)$, o que acontece quando o valor que se deseja inserir coincide com o valor da raiz. No caso médio, conhecidamente temos que o custo é $O(\log n)$. E por fim, no pior caso, ou seja, quando a árvore se degenera a ponto de ser equivalente a uma lista, o custo é $O(n)$.

Por fim temos os métodos accessTree() e convetToList(). Ambos os métodos tem comportamento similar, fazendo algo com o nó atual e chamando a função de forma recursiva para seus filhos. O algo que é feito no método accessTree() é apenas registrar no log de acesso, $O(1)$, e no convertToList() é inserir em uma lista estática, $O(1)$. Sabendo disso, temos que a função de custo dessas operações fica $T(n) = 2T(n/2)+1$, portanto, pelo Teorema Mestre, o custo assintótico é de $O(n)$.

O método destrutor da lista tem custo $O(n)$, pois apenas destrói a raiz da árvore, o que tem custo $O(n)$.

d- Lista:

Os métodos construtores e set da lista tem complexidade da ordem $O(1)$, pois apenas inicializam valores, sem utilizarem nenhum loop ou chamada recursiva para tal.

O método de inserção na lista, assim como os citados acima, também tem custo $O(1)$, porém o motivo é diferente. As inserções são realizadas na última posição do vetor, portanto, para não precisar percorrer o vetor até encontrar o primeiro valor não inicializado, o atributo conter foi criado na classe lista, indicando tanto o índice da posição que o novo elemento deve ser inserido quanto a quantidade de elementos já inicializados.

Em seguida temos o método accessList(), que tem como única função registrar todos os elementos da lista no log de acesso. Desta forma, ele consiste em um loop simples com passo de tamanho 1 e que passa por todos os elementos da lista. Sendo assim, temos que o custo desse método é O(n), sendo n o número de elementos da lista.

Sendo os mais importantes do programa e o objetivo deste trabalho, temos agora os métodos de ordenação. O primeiro que temos é o método de seleção, que encontra o menor elemento da lista, e o troca com a da primeira posição, em seguida, encontra o segundo menor e o coloca na segunda posição, e assim por diante. O resultado disso é um uma função de custo da ordem de $O(n^2)$.

O segundo método de ordenação usado, sendo este obrigatório, foi o QuickSort, que vai particionado o vetor em partições cada vez menores, dividindo a partição entre valores maiores que o pivô e valores menores que o pivô, e fazendo o mesmo com essas partições até que se tenha o tamanho 1 ou 0. Tendo divido todas as partições entre aquelas de tamanho 1 e 0, basta juntá-las de modo que a partição menor fique antes do pivô e a maior depois dele, para obter um vetor ordenado. Sabendo então do funcionamento desse método, observamos que, no melhor caso ele tem custo O(n), no calor médio O(n log n), e, no pior caso $O(n^2)$.

Um problema do QuickSort é a recursão que ele utiliza, sendo essa cara, pois a cada chamada é necessário empilhar uma nova função na pilha de execução. Como forma de amenizar este problema, é definido um tamanho mínimo que uma partição, e, caso uma partição tenha um tamanho menor do que esse mínimo, esta é ordenada de acordo com algum outro método simples que não utiliza de chamadas recursivas, este caso, o método de seleção.

Outro problema do QuickSort é o seu pior caso, acontece quando o vetor está inversamente ordenado e escolhe-se sempre o primeiro valor como sendo o pivô, portanto, como forma de evitar escolher sempre o maior ou menor valor como sendo o pivô, são amostrados n valores do vetor a ser ordenado, e esses n valores são ordenados. Após ordenados, é escolhido a mediana destes valores, e essa será usada como pivô para realizar o QuickSort, o que garante que o pior caso não acontecerá quando não houver repetição de elementos, que é o caso deste programa. O método utilizado para ordenar os candidatos a pivô foi a seleção, de modo que seja acrescentado um custo de $O(n^2)$, para n menor que o tamanho do vetor.

e- Árvore vs lista como estrutura para inserção:

Tendo em vista que uma lista estática não é uma estrutura adequada para se armazenar a entrada deste programa, pois essa pode ser arbitrariamente grande, de modo que o número de posições inutilizadas que poderiam surgir ocuparia uma quantidade desnecessária de memória. Desta forma, uma lista encadeada deveria ser usada para se armazenar os valores.

Sabemos também, que o custo de se realizar uma busca em uma lista encadeada, tem caso médio como sendo $T(n) = n/2$, ou seja, O(n). Isso que, a inserção de n palavras em na lista teria custo de $O(n^2)$ no caso médio.

Diferentemente das lista, no caso médio, o custo de busca em uma árvore binária é da ordem de O(log n). Portanto, o custo média da inserção de n palavras na árvore é da ordem de O(n log n), enquanto, o custo máximo, é da ordem de $O(n^2)$, o caso média da lista encadeada. Portanto, no caso médio, trabalhar com uma árvore tem custo menor do que trabalhar com uma lista.

Porém, surge um problema ao se escolher uma árvore sobre uma lista encadeada, que é necessária a realização de uma conversão de árvore para lista de forma que a ordenação possa ser feita. Sendo este um custo extra quando comparado com a lista. Portanto, observando do ponto de vista assintótico, esse custo extra ainda é irrelevante, pois, no caso médio, o inserção e a conversão de uma árvore tem custo de O(n log n) + O(n) = O(n log n), que ainda é menor do que o caso médio da lista.

Desta forma, concluímos que, no caso médio, o custo total do programa, utilizando uma árvore e uma lista, é de O(n log n) + O(n) + O(n log n) + O(n) = O(n log n), enquanto o uso de somente uma lista, no caso médio, custa $O(n^2)$ + O(n log n) + O(n) = $O(n^2)$. E no pior caso, ambos têm o custo de $O(n^2)$.

Concluímos assim que, a escolha de se utilizar uma árvore e depois convertê-la em uma lista é parte do compromisso da ciência da computação de balancear memória e tempo, de modo que, mesmo gastando mais memória e precisando de uma operação a mais, o uso da árvore faz com que, no caso média, o programa consiga executar de forma mais rápida do que executaria somente com uma lista.

4- Estratégias de Robustez:

Como estratégias de robustez, foram utilizadas asserções que interrompem o programa. O motivo da escolha da interrupção do programa se deve ao tanto a resultados errados que surgiriam devido aos problemas quanto a outros real impossibilidade de continuar-se executando o programa.

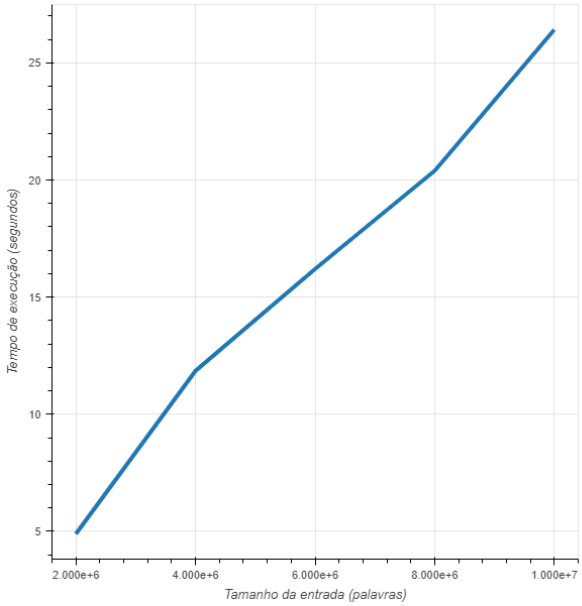
As asserções foram utilizadas para verificar se as entradas fornecidas pelo usuário são válidas, dentre essas estão: (1) a quantidade de candidatos a pivô deve ser sempre maior que 0, (2) a menor partição para se executar o método de otimização do QuickSort deve ser maior que 1 elemento, (3) os arquivos de entrada e saída devem poder serem abertos pelo programa.

5- Análise Experimental:

Nesta seção serão detalhados os experimentos feitos a respeito da análise de localidade de referência do programa e o a análise de tempo de execução do programa.

a- Tempo de Execução:

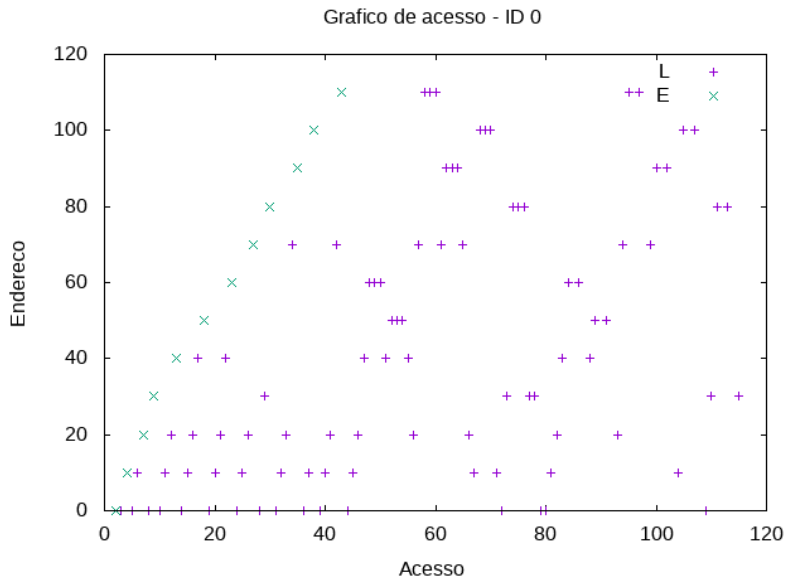
Pata a análise de tempo de execução do programa, foi criado um script python que gera textos contendo palavras aleatórias. Os textos gerados tinham os tamanhos de 2 milhões de palavras, 4 milhões, 6 milhões, 8 milhões e 10 milhões. Os resultados obtidos estão plotados no gráfico da página seguinte.



Esse gráfico evidencia fortemente que a escolha de se utilizar uma árvore foi uma boa ideia, tendo em vista que apresenta um comportamento quase linear com o aumento da quantidade de palavras. Ou seja, o programa é mais escalonável do que seria caso utiliza-se apenas uma lista, perdendo para o uso de apenas uma lista para entradas pequenas, mas trazendo enormes benefícios para grandes entradas.

b- Análise de Localidade de Referência:

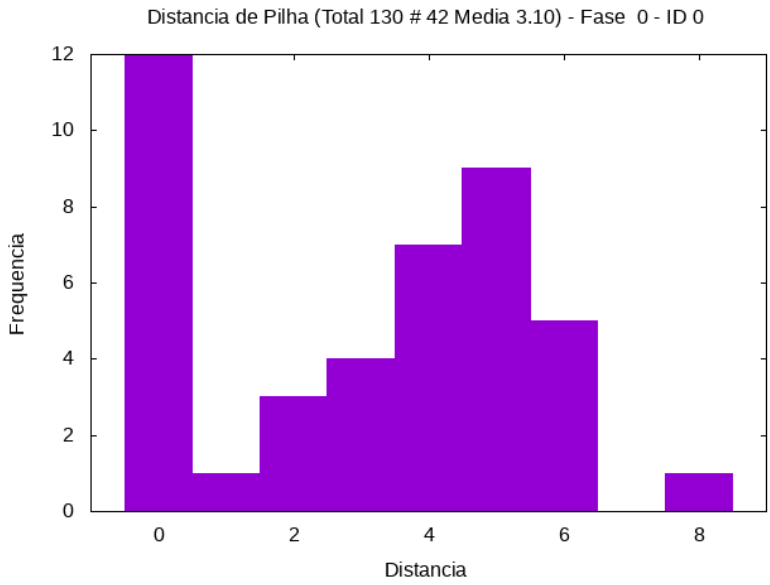
Para analisar a localidade de referência do programa devemos observar suas estruturas de forma separada, primeiro observaremos a árvore e em seguida a lista. Para esse teste foi utilizado um texto com 12 palavras, de modo a evitar que a imagem ficasse muito poluída e difícil de compreender.

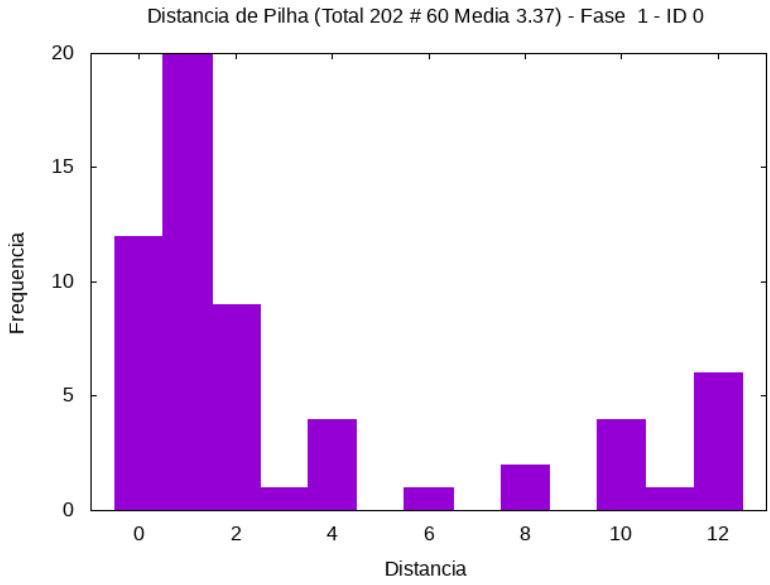


Esse mapa é referente á árvore de busca binária que fui utilizada para armazenar a entrada. Na primeira fase do programa, que se finaliza com a última cruz verde, observamos o processo de inserção de novas palavras. Os saltos verticais entre os pontos

roxos durantes essa fase se deve ao fato de que, uma posição anterior é acessada e, em seguida, há um salto dessa posição para seu filho, ou da direita ou da esquerda, sendo esse salta cada vez maior à medida que se vai caminhando pela árvore.

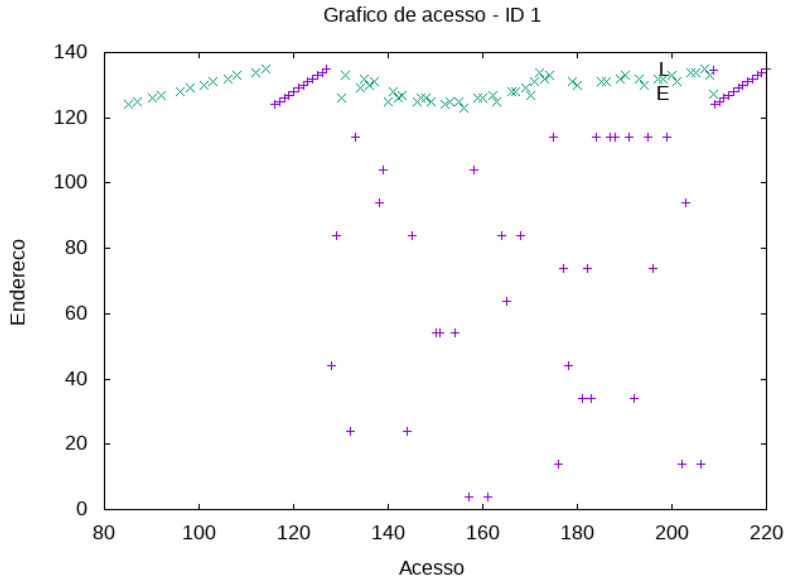
A segunda fase pode ser dividida em duas partes: a arte de acesso da árvore e a de conversão da árvore. Durantes a parte de acesso, podemos observar saltos similares aos da primeira, porém, quando chegam em uma determinada altura, uma mesma posição é acessada de forma seguida, e depois há uma queda para a posição anterior. Isso caracteriza que o caminhamento alcançou uma folha, voltando assim para o nó anterior e indo para o próximo. Um comportamento idêntico a esse pode ser observado na segunda parte da segunda fase, tendo em vista que fazem o mesmo percurso de acessos.





Os histogramas de distância de pilha da árvore fazem sentido em apresentarem o grau de dispersão que apresentam, tendo em vista que os acessos precisam ser intermediados pelos nós pais.

Finalizando a análise de localidade de memória da árvore, partimos agora para a lista.



Assim como na árvore, a parte do vetor pode ser dividida em duas, a primeira na qual ele recebe os valores vindos da árvore e a segunda que compreende o QuickSort e a impressão.

Observando o momento das inserções na lista, percebemos que escrever no vetor vai do acesso 80 até o 120, enquanto o comportamento esperava-se que ele se ocupa apenas 12 acessos, porém, parando pra pensar, podemos observar um espaçamento no eixo dos acessos, e esse espaçamento não é uniforme para todas as escritas. Isso acontece pois os valores na árvore não são acessados de forma contínua, mas valores que já foram acessados são acessados novamente. Momentos com maior espaçamento indicam que acabou de ser escrito um valor que estava presente em uma folha, e que o cursor regrediu em mais de um nível antes de escrever novamente.

Analisando o comportamento da segunda parte, percebemos ele como sendo meio caótico e desordenado, porém, isso só ocorre porque a plotagem desses valores ficou muito concentrada na parte de cima. Se observarmos atentamente aproximadamente na posição 130, vemos duas escritas com um espaçamento entre seus endereços, as próximas duas escritas têm um espaçamento menor ainda, e uma terceira dupla ainda menor. Isso se dá devido ao método de particionar o vetor, que coloca um cursor na primeira posição que percorre o vetor em ordem crescente e outro que começa na última e vai em ordem decrescente, e, caso as condições sejam satisfeitas, uma troca é feita, sendo então registradas as escritas.

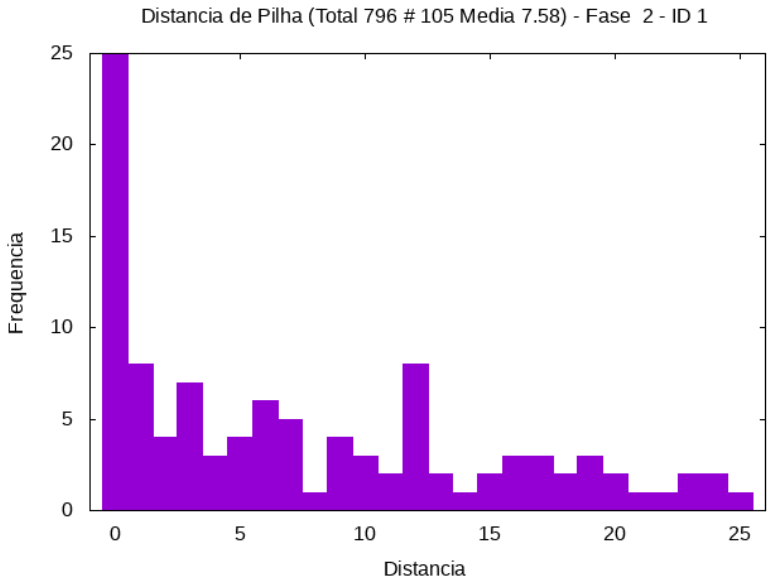
Observamos também que, logo assim que acaba o primeiro particionamento, as escritas se concentram abaixo do endereço onde o fenômeno anterior chegou na convergência, até aproximadamente no acesso 170, onde o oposto ocorre e as escritas se concentram acima desse ponto.

Observando ainda mais atentamente o que foi dito no parágrafo anterior, observamos percebemos que a parte abaixo da linha de convergência replica o

comportamento descrito, e o mesmo ocorre com a parte acima. E isso continua acontecendo sendo observado a medida em frações cada vez menores.

Esse comportamento pode ser atribuído ao caráter recursivo do QuickSort, que gera uma partição que vai do primeiro elemento até o elemento que foi descrito anteriormente como ponto de convergência, e outra que cobre o resto do vetor. Obtendo essas duas partições, a primeira, que pode ser vista graficamente como a parte abaixo da linha de convergência, passa pelo mesmo processo que o vetor acabou de passar, sendo particionado em dois novamente. E, a primeira parte, a segunda é executada, justiçando assim os saltos que ocorrem entre a linha de convergência.

Por fim é impresso o vetor, sendo a reta de roxo ao final do esquema.



O histograma de distância de pilha da lista, assim como na árvore, apresenta um comportamento disperso. Isso ocorre pelos acessos estarem sendo feitos simultaneamente de frente para trás quando de trás para frente.

6- Conclusão:

Por fim, temos como resultado um programa robusto, com um bom desempenho e que conseguirá lidar com entradas grandes de forma mais assertiva do que programas mais simples. Porém, o sacrifício feito para obter isso altas distâncias de pilha e grande dispersão dos acessos no mapa de acessos.

Para finalizar então o trabalho, reitero o compromisso presente na computação de balancear memória e velocidade. Sempre há um custo em se aumentar a velocidade, assim como existe um custo em poupar memória, e acredito ter feito um bom trabalho balanceado os dois neste projeto.

7- Bibliografia:

Slides: Estruturas de Dados, Ordenação: QuickSort, Professores: Luiz Chaimowicz e Raquel Prates.

8- Apêndice A: Instruções para compilação e execução:

a- Compilação:

A compilação é simples, basta entrar no diretório raiz do projeto e utilizar o comando make. O arquivo makefile irá se responsabilizar por criar os arquivos compilar os arquivos e gerar um executável.

b- Execução:

Para executar o programa a partir do diretório raiz:

- bin/tp2.exe

seguido pelas flags:

- -[i | I] <caminho_do_arquivo_de_entrada> : obrigatório, informa o caminho do arquivo de entrada
- -[o | O] <caminho_do_arquivo_de_saída> : obrigatório, informa o caminho do arquivo de saída
- -[s | S] <inteiro> : opcional, informa o tamanho mínimo que o QuickSort deve particionar
- -[m | M] <inteiro> : opcional, informa a quantidade de candidatos a pivô a serem analisados
- -[l | L] : opcional, habilita registro de localidade de memória