Trabalho Prático 2

Gabriel Franco Jallais

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais(UFMG)

Belo Horizonte – MG – Brasil

gfjallais@ufmg.br

Entrega: 04/07/2022

1. Introdução

O problema proposto para este trabalho foi contar o número de ocorrências de palavras em um arquivo de texto, sendo que palavras com diferenças de caixa alta ou baixa, mas com os mesmos caracteres, devem ser contabilizadas juntas, como por exemplo, “BanaNa” e “bAnAnA”, ambas devem aparecer no arquivo de saída como “banana” e junto a ela o número de ocorrências no texto. Para a escrita no arquivo de saída, outro problema foi proposto, imprimir a saída com base em uma ordem lexicográfica diferente do padrão, alfabética, esta ordem é passada junto ao texto no arquivo de entrada. Essa ordenação deve ser realizada utilizando o algoritmo de ordenação *Quick sort*, com otimizações.

Para tal tarefa, foram criadas diferentes classes e funções, para tratar as palavras e armazená-las, além de funções para realizar a comparação com a nova ordem lexicográfica e realizar a ordenação.

Importante ressaltar que a entrada é passada como parâmetro é passada como parâmetro, assim como o arquivo de saída, o número de elementos a serem utilizados para realizar a otimização da mediana de n números para definir o pivot das partições do Quicksort, e o tamanho máximo das partições para que se use um algoritmo simples, no meu caso Selection sort para realizar a ordenação da partição. O arquivo de entrada contém dois blocos, #TEXTO e #ORDEM, que, em seguida deles, aparecem, respectivamente, as palavras do texto em quantidade indefinida, e a ordem lexicográfica a ser seguida, uma linha com 26 caracteres.

Nas próximas seções serão dados mais detalhes quanto a implementação, a análise de complexidade das operações, além de quais foram as estratégias de robustez adotadas, quais testes foram feitos, junto a análise experimental, por fim, é apresenta a conclusão, a bibliografia utilizada e instruções para compilação e execução.

2. Implementação

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

Foi compilado e executado no sistema operacional Ubuntu 20.04 dentro do WSL2 no Windows 11, com um processador i7-1165G7 com 16 GB de memória RAM.

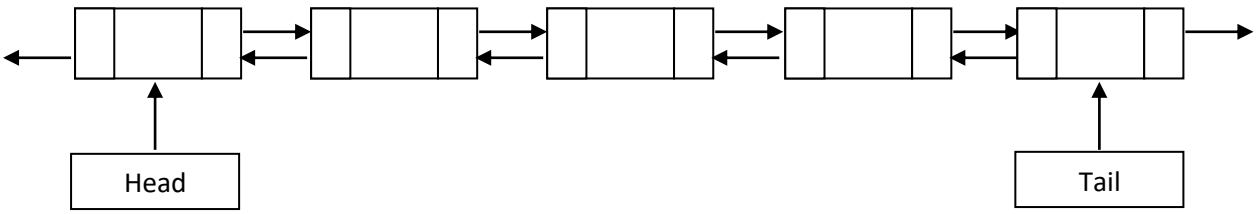
2.1 Estrutura de Dados

A implementação do programa teve como base a estrutura de dados de uma lista duplamente encadeada, para armazenar as palavras do texto e efetuar a contabilização das suas respectivas ocorrências. Esta foi escolhida em lugar da simplesmente encadeada, pois possui metade do custo de acesso por posição e custo constante para inserção ao final.

Foi utilizado um template para a sua implementação, permitindo a sua reutilização em outros cenários.

Para otimizar um pouco os custos assintóticos, a lista foi implementada com uma célula cabeça e com um atributo que armazena seu tamanho, além disso, cada node armazena um item do tipo T, no caso string, e um inteiro que contabiliza o número de ocorrências desse item, caso, durante a inserção, tenha sido passado um parâmetro que solicita a contagem de ocorrências.

Segue um esquema da estrutura de dados:



2.1.1 Implementação da Lista Duplamente Encadeada e Node

Node: Possui como atributos: um ponteiro para o próximo node e outro para o anterior, um item da classe T e um inteiro quantidade, que guarda a quantidade de ocorrências do item.

Atributos: *next, *prev, item, quant.

A classe node possui como métodos apenas seu inicializador, que atribui NULL a next e prev e 1 a quantidade, q_increment() que incrementa o atributo quantidade, get_item, get_quant, get_next, get_prev que retornam os valores de item, quant, next e prev, respectivamente, e set_node(T item, int quant), que altera os valores do node.

List: Possui como atributos: um ponteiro para o primeiro e para o último node da lista, e um inteiro size que armazena seu tamanho.

Atributos: *head, *tail, tam.

A classe lista possui como métodos:

lista(): Inicializa head com um new node(), atribui a tail o valor de head e inicializa size com 0.

~lista(): Itera pela lista desalocando cada node.

node* get_head(): Retorna o ponteiro para a cabeça da lista, no caso, retorna o atributo head.

node* get_tail(): Retorna o ponteiro para o último node da lista, no caso, retorna o atributo tail.

carta get_item(int** pos**): Retorna o item na posição *pos* da lista.

void insert(T** item, **bool** count_duplicates**): Caso *count_duplicates* seja false, insere o item passado na lista, caso seja true, antes de inserir, busca o item na lista, se encontrado apenas incrementa a quantidade deste e não insere nada na lista, caso contrário insere o item na lista.

node* search(T *****item**): Pesquisa pelo item na lista.

node* at_position(int** pos**, bool** before**): Retorna um ponteiro para o node na posição *pos*, caso *antes* seja *false*, e para o node anterior a *pos*, caso *antes* seja *true*.

int get_size(): Retorna o tamanho da lista.

A implementação desta classe se encontra no arquivo list_template.h.

2.2 Funções

Foram implementadas funções para processar as palavras do texto de entrada, para que elas sejam armazenadas com todas as letras em caixa baixa e sem nenhum sinal de pontuação (“,”, “.”, “!”, “?”, “:”, “;” e “_”) e assim o resultado seja como o esperado. Ademais, também foram formuladas funções para a ordenação da lista após a leitura da entrada, para que esta seja impressa ordenada com a nova ordem lexicográfica.

2.2.1 process_string.h

Dentro desse arquivo header encontram-se duas funções, *check_prohibted_chars(char c)* e *process_string(std::string *str)*. A primeira é responsável por checar se o char passado como parâmetro da função pertence ao conjunto de pontuações especificado no início da seção. Já a segunda, itera pela string, checa se os seus caracteres são válidos e se este for o caso os coloca em caixa baixa, caso seja encontrado algum char inválido ele é removido da string.

2.2.2 selection_sort.h

Neste arquivo encontram-se as funções *compare_given_order(std::string str1, std::string str2, int* order)*, *swap(node<std::sting> *xp, node<std::string> *yp)* e *selection_sort(list<std::string> *list, int start, int end, int *order)*. A primeira recebe duas strings, e um vetor de inteiros order, compara as strings caractere a caractere e retorna -1 se a

str1 for “menor” que a str2, isto é, ela vem antes na ordenação, 1 se str1 for “maior” e 0 se ambas forem iguais.

O vetor order é criado na leitura do arquivo, ele possui 26 posições, sendo cada posição uma letra do alfabeto na ordem alfabética. Cada letra(posição) recebe o valor da posição em que ela aparece no arquivo de entrada incrementado em 97 unidades, de forma a mapear a nova ordem no intervalo de valores da tabela ASCII em que os caracteres aparecem, permitindo que a comparação seja feita com caracteres que não sejam letras.

Dada a ordem: B C D P F G H J K L M N A Q R S T U O X Z W Y E I V

Order: A(109) B(97) C(98) D(99) E(120) F(101) G(102) H(103) I(121) J(104) K(105) L(106) M(107) N(108) O(115) P(100) Q(110) R(111) S(112) T(113) U(114) V(122) W(118) X(116) Y(119) Z(117)

Já a segunda função, *swap(node<std::sting> *xp, node<std::string> *yp)*, apenas faz a troca dos valores dos nodes passados, quantidade e item armazenado, no caso uma palavra.

Por fim, *selection_sort(list<std::string> *list, int start, int end, int *order),* aplica o algoritmo de ordenação selection sort em um intervalo da lista passado, delimitado por star e end, e para fazer as devidas comparações utiliza o vetor order e a função *compare_given_order(std::string str1, std::string str2, int* order)*, e para a troca de elementos a função *swap(node<std::sting> *xp, node<std::string> *yp)*.

2.2.2 quicksort.h

Neste arquivo encontram-se apenas as funções *partition(list<std::string> *list_p, int left, int right, int *i, int *j, int m, int* order)* e *quicksort(list<std::string> *list, int left, int right, int m, int s, int* order)*, sendo a primeira utilizada para realizar as partições da lista, assim como mostrado em sala, além de determinar o pivot com base na mediana de m elementos, sendo m um inteiro passado como argumento na execução do programa (caso a partição o tamanho da partição seja menor do que m, foi utilizada a mediana de 1 elemento, no caso o pivot selecionado é o elemento do meio da partição), já o segundo executa o algoritmo quicksort de maneira recursiva, e caso o tamanho do intervalo passado para a ordenação seja menor ou igual a s, a função selection_sort é chamada para ordenar este intervalo da lista.

3. Análise de Complexidade

Para a análise de complexidade do programa, serão levadas em conta apenas as funções referentes às operações de ordenação, acesso e pesquisa na lista duplamente encadeada e execução do programa main.cpp, visto que estas variam em função do número de palavras n no arquivo de entrada. Será feita apenas a análise de complexidade em relação ao tempo de execução, visto que até então, a análise de complexidade de espaço não foi tão bem abordada em sala.

*- void selection_sort(list<std::string> *list, int start, int end, int *order)*

Como visto em sala, e como é facilmente provável, o algoritmo de ordenação selection sort, possui complexidade assintótica $\boldsymbol{O(n^2)}$, em todos os casos, pior, médio e

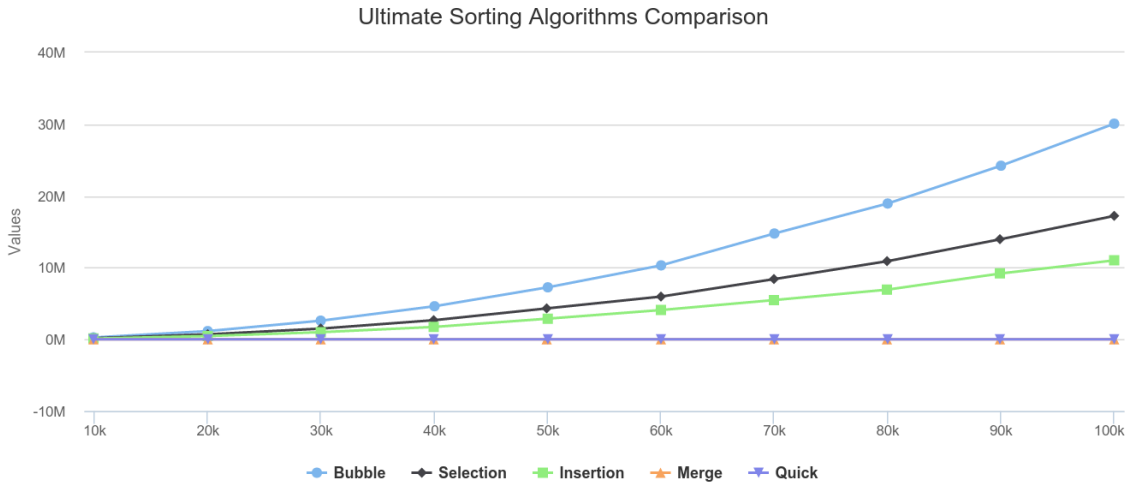
melhor caso, onde n é o número de nodes da lista, no caso o número de palavras únicas no arquivo e entrada.

- *void partition(list<std::string> *list_p, int left, int right, int *i, int *j, int m, int* order)*

A função responsável pela partição iterara por todo o segmento da lista a cada execução, ademais, a depender do tamanho de m, será executado um loop com m iterações para adicionar elementos a uma lista, sendo o custo de inserção constante, e após isso, selection_sort será executado sobre esta nova lista para determinar o pivot, portanto, é fácil concluir que a complexidade assintótica desta função é $O(m^2 + n)$, onde m é o tamanho de m e n é o tamanho do intervalo da lista no qual se deseja realizar o particionamento.

- *void quicksort(list<std::string> *list, int left, int right, int m, int s, int* order)*

A função em questão executa apenas as chamadas recursivas do algoritmo e checa o tamanho da partição para, se for o caso, chamar a função selection_sort, como visto em sala essas chamadas recursivas serão feitas log n vezes no melhor caso e no caso médio, e custo de partition é $O(m^2 + n)$, como visto, ou seja, ele possui complexidade $O((m^2 + n)*log\ n)$ na maioria dos casos. Analisar a complexidade dessa função levando em consideração a chamada do algoritmo selection_sort é uma tarefa complicada, visto que quanto maior for o parâmetro s, a complexidade do quicksort se tornará mais próxima da complexidade do selection sort, contudo, quando s é pequeno o suficiente, o cust-benefício dessa chama compensa. No caso do selection sort, analisando assintoticamente, como o seu custo é sempre $O(n^2)$ mesmo no pior caso do quick sort, o máximo que vai ocorrer é ambos terem o mesmo custo de tempo, visto que o pior caso do quick sort é também $O(n^2)$, a diferença estaria no custo de memória, visto as chamadas recursivas do quick sort.



Na prática o que se pode encontrar é o seguinte:

Fonte: Bubble, Selection, Insertion, Merge, Quick Sort Compared (titrias.com)

Até uma entrada de tamanho 30.000 o tempo de execução do quick sort e do selection sort são bem similares, e como dito anteriormente o custo de pilha de execução será menor para o selection sort.

- *node<T>* list<T>::search(T *item)*

Para realizar a busca na lista duplamente encadeada o procedimento é o mesmo que a busca em uma lista simplesmente encadeada, pois, visto que não há ordenação prévia garantida, indexação o hashing, é necessário percorrer toda a lista para encontrar o item. Portanto o seu custo assintótico é ***O(n)***.

- *node<T>* list<T>::at_position(int pos, bool before)*

O acesso, todavia, pode ser feito de maneira mais eficiente do que em uma lista simplesmente encadeada, primeiramente checa-se o valor do parâmetro pos passado, se este for menor que a metade do tamanho da lista, percorre-se a lista começando pelo lado esquerdo, caso contrário, pelo lado direito, sendo assim, sua complexidade é ***O(n/2)***.

- *A execução do arquivo main.cpp*

As funções responsáveis para pegar os argumentos de execução do programa, para abrir os arquivos de entrada, saída e log, possuem custo constante, logo praticamente irrelevantes para a análise, a leitura do arquivo, no entanto possui custo ***O(n)***, onde n é o número de palavras no arquivo. Ademais há a execução da função quick sort ***O((m² + n)*log n)***, para ordenar a lista para a impressão, e em seguida a impressão no arquivo de saída com custo ***O(n)***, logo, o custo da execução do programa main.cpp é ***O((m² + n)*log n)***.

4. Estratégias de Robustez

Para garantir a boa operação do programa, isto é, que ele seja robusto para tolerar erros do usuário e se defender de tentativas de "ataque" que visam achar falhas no programa para explorá-las, foram tomadas as seguintes estratégias.

Primeiramente, é feita a leitura dos argumentos para a execução e checa se eles são válidos para que os arquivos abram corretamente e as operações sejam executadas da maneira certa, para isso, são usadas as funções *parseArgs(int argc,char ** argv)*, o método open da classe fstream e a função *erroAssert(e,m)* da biblioteca msgassert.h.

Ademais, para cada função/operação descrita na seção 2, são feitas checagens para que as entradas cumpram os requisitos para as operações requisitadas, isto é, no caso dos métodos posiciona, get_item, set_item da classe lista, faz-se a checagem do valor do parâmetro pos passado, para que não haja a tentativa de acesso a posições inexistentes na lista, e, consequentemente, um erro de segmentation fault seja gerado.

Além disso, nas funções de ordenação checa-se se o intervalo solicitado para a ordenação da lista é menor do que o tamanho da lista, para que comportamentos

inesperados não ocorram, bem como acessos a posições inexistentes na lista, mas os métodos da lista também garantem a segurança.

5. Testes

Para analisar desempenho computacional do programa, além de fazer a análise do padrão de acesso à memória e localidade de referência, foram feitos testes utilizando um gerador de texto lorem ipsum (https://www.lipsum.com/):

Para a análise de desempenho, a análise de memória não foi ativada, para melhor fidelidade e representação do desempenho do programa, e foram dadas entradas de 1000 a 10000 palavras indo de 500 em 500, com m = 5 e s =10.

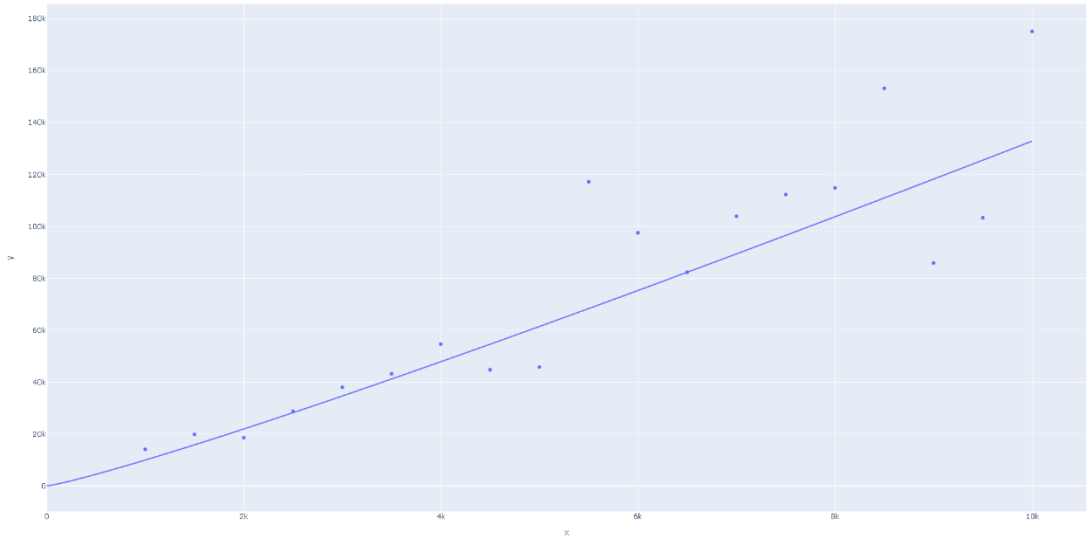
Já para a análise de acesso e localidade de memória, foi utilizada uma entrada pequena, com apenas 65 palavras, o arquivo 1.tst.i, passado como exemplo de input junto ao enunciado do trabalho.

6. Análise Experimental

Para efetuar a análise do programa em questão foram utilizados os softwares memlog e analisamem, que foram responsáveis por mapear o tempo de execução e o acesso à memória.

6.1 Desempenho Computacional

Para verificar a análise de complexidade feita previamente, foi feita uma sequência de testes e traçada uma curva para capturar o comportamento do programa em função do aumento da entrada, segue o gráfico:



Os pontos foram representam as amostras obtidas dos testes com textos gerados aleatoriamente, o tempo de execução teve sua escala alterada para que fosse possível visualizar a curva. Já a curva é dada pela função y = x*logx, ou seja, a complexidade prevista para o programa, visto que m e s foram suficientemente pequenos.

Os resultados para cada entrada são oscilantes devido ao hardware e às outras atividades que estavam sendo executadas em paralelo no computador, levando isso em consideração foram feitos alguns testes para tentar melhorar a precisão da análise.

6.1.1 Resultados do gprof

Para analisar os resultados do gprof, utilizaremos uma entrada com 212036 palavras, para haver uma visualização mais direta do funcionamento do programa e para que o tempo de execução seja grande o suficiente para gerar resultados visíveis.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
81.07	14.70	14.70				list<std::string>::at_position(int, bool)
10.11	16.53	1.83				std::operator==<char>
6.19	17.65	1.12				list<std::string>::insert(std::string, bool)
0.94	17.82	0.17				node<std::string>::node()
0.80	17.97	0.15				std::compare(char const*, char const*, unsigned long)
0.39	18.04	0.07				compare_given_order(std::string, std::string, int*)
0.17	18.07	0.03				process_string(std::string)
0.14	18.09	0.03				_GLOBAL__sub_I_ml
0.11	18.11	0.02				partition(list<std::string>*, int, int, int*, int*, int, int*)
0.06	18.12	0.01				selection_sort(list<std::string>*, int, int, int*)
0.06	18.13	0.01				swap(node<std::string>*, node<std::string>*)
0.06	18.14	0.01				node<std::string>::q_increment()
0.03	18.15	0.01				node<std::string>::get_item()
0.03	18.15	0.01				node<std::string>::get_quant()

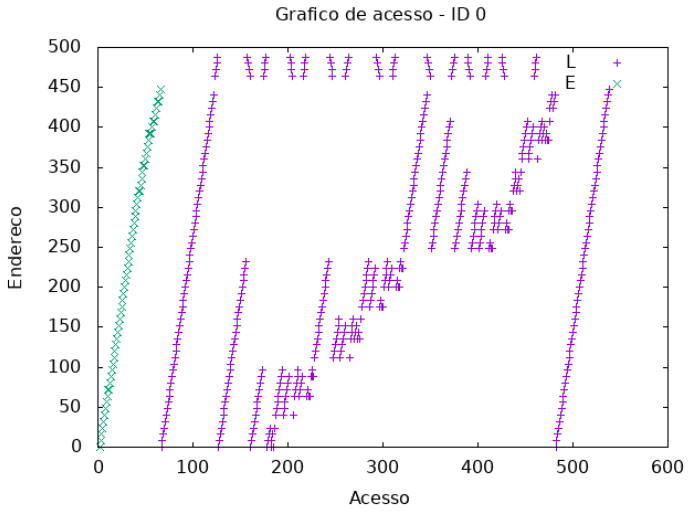
***Alguns dados foram retirados ou tiveram seus nomes reduzidos do resultado para fins de apresentação, como é o caso das operações que envolvem regex, com elas no resultado seria impossível conte-las no espaço requisitado para a documentação, além da interpretação extremamente dificultada. Mantiveram-se todas as funções referentes ao programa em si, citadas em seções anteriores.**

Vê-se como o método at_position da list é o que mais consome processamento e tempo no programa, visto que é por meio dela que se faz o acesso aos elementos da lista nos algoritmos de ordenação e na função main, por se tratar de um texto extenso, a busca linear, como aplicada no método, é bem custosa, por isso o resultado obtido.

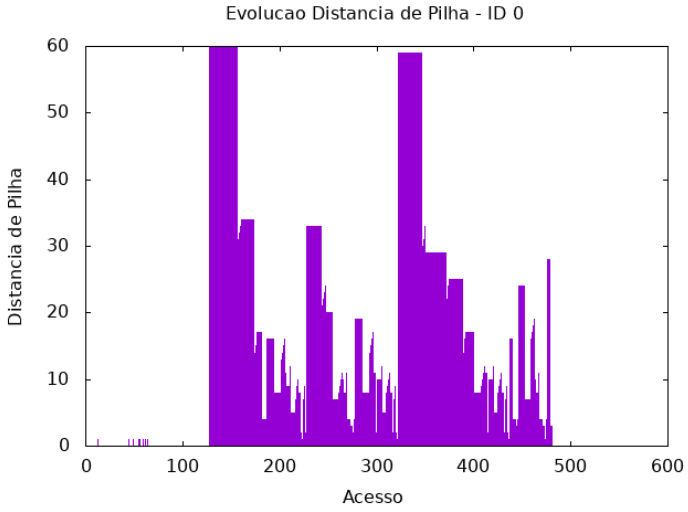
6.2 Análise de Padrão de Acesso à Memória e Localidade de Referência

Agora, para entender melhor como o programa acessa memória e a sua localidade de referência, observaremos os gráficos gerados por meio do software gnuplot, e cujos dados foram fornecidos pelo programa analisamem, utilizando um texto com 65 palavras, para uma visualização menos poluída.

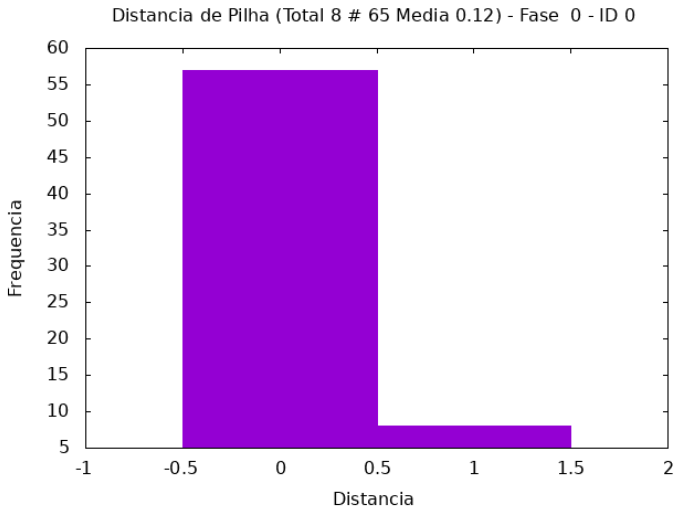
Primeiramente, cabe dizer que, o id 0 é referente a lista de palavras. Os gráficos mostram como foi feito o acesso à memória ao decorrer do tempo, pontos (+) próximos no eixo x indicam endereços que foram acessados em intervalos próximos de tempo.



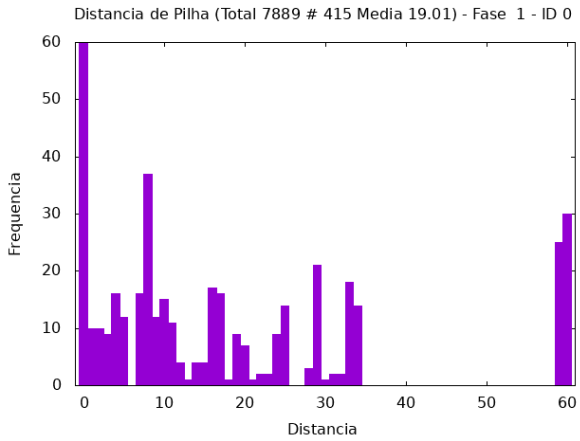
O programa main.cpp segue os seguintes passos: leitura do arquivo de entrada, em seguida, a criação da lista de palavras, que pode ser vista pelos pontos (x) verdes no lado esquerdo do gráfico, após isso, é feita a ordenação, o que justifica o acesso um pouco caótico e segmentado, em duas partes, divididas no ponto 300, por fim é feita a impressão no arquivo de saída, a faixa linear mais a direita, passando por todos os itens da lista.



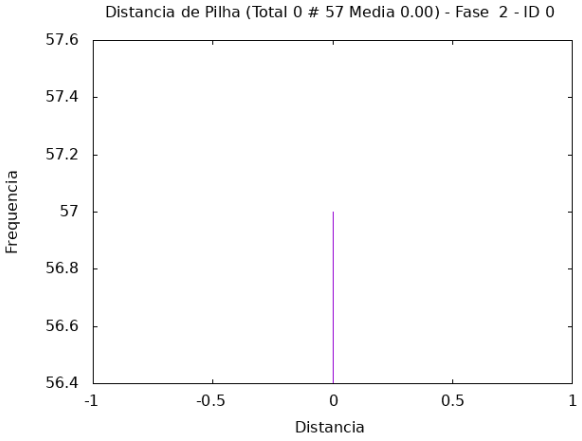
Da mesma forma observa-se como o gráfico de distância de pilha é afetado, os retângulos de altura 60, representam a criação e a impressão da lista, já os demais, de diversos tamanhos, representam os demais acessos feitos durante a ordenação.



A primeira fase, fase 0, engloba apenas a criação da lista, por isso o histograma é mais uniforme.



Já a segunda, captura o comportamento do quicksort, é aí que vemos o comportamento caótico visto nos gráficos de acesso e de evolução da distância de pilha, a depender dos pivots e das partições a distância de pilha se comporta de forma diferente.



Por fim, na terceira fase, é feita apenas a impressão dos itens da lista no arquivo de saída, por essa razão, observa-se o gráfico acima, o acesso não é contíguo assim como um vetor estático, dessa forma todos os acessos são feitos com uma distância de pilha igual a 0.

7. Conclusões

O trabalho consistiu em contabilizar o número de ocorrências de palavras, independente de diferenças de caixa alta e baixa ou pontuação, e imprimi-las no arquivo de saída, junto ao número de ocorrências, de forma ordenada, com base em uma nova ordem lexicográfica passada no arquivo de entrada, utilizando o algoritmo quick sort com otimizações. Para tal tarefa utilizou-se os conhecimentos obtidos em sala, além realizar a análise do desempenho em relação ao tempo de execução e o acesso à memória conforme o tamanho da entrada cresce, nesse caso, o número de palavras no arquivo de entrada.

Por se tratar de uma tarefa relativamente simples, não houve grandes dificuldades para a sua realização, sendo a tarefa de análise e adaptação do quick sort as mais laboriosas. Pôde-se observar como as funções seguem a análise assintótica feita na seção 2, e como o acesso a memória e a distância de pilha foram coerentes com a implementação feita para as operações e coerentes com as entradas fornecidas.

8. Bibliografia

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Algoritmos: Teoria e Prática. 3a edição. Elsevier, 2012. ISBN 9788535236996

Nivio Ziviani. Projeto de Algoritmos com implementações em Pascal e C. 3ª edição. Cengage Learning, 2011. ISBN 9788522110506

Apêndice

Instruções para compilação e execução

Para facilitar a compilação e execução do programa main.cpp, foi disponibilizado um arquivo Makefile com comandos mais simples.

Para compilar o programa, basta possuir o arquivo Makefile no mesmo diretório que os outros programas, estruturados de forma:

|- src
|- bin
|- obj
|- include
Makefile
entrada.txt

Onde src deve possuir os arquivos .cpp e include deve possuir os arquivos .h e executar o comando “make comp”.

Para executar o programa basta executar o comando na linha de comando no seguinte formato:

(EXE) (COMANDO OPICIONAL -p) (SAÍDA DA ANÁLISE DE DESEMPENHO) (COMANDO OPICIONAL -l) -[i|I] (ARQUIVO DE ENTRADA) -[o|O] (ARQUIVO DE SAÍDA) –[m|M] (VALOR PARA MEDIANA DE M NÚMEROS) –[s|S] (VALOR MÁXIMO PARA O USO DE UM ALGORITMO SIMPLES PARA A ORDENAÇÃO NO QUICKSORT)

EXE = tp1
COMANDO OPICIONAL 1 = -p, para a geração de arquivo .out com dados de desempenho computacional
COMANDO OPICIONAL 2 = -l, para a geração de mapas de acesso a memória
ARQUIVOS DE ENTREDA = entrada.txt

Para realizar as experimentações vistas na documentação basta executar o comando “make mem”, para gerar os mapas de acesso e histogramas, o comando “make perf”, para gerar arquivos .out para a análise de desempenho, e o comando “make gprof”, para gerar arquivos .gprof para a análise de desempenho. ***É necessário possuir o programa analisamen.c**

Por fim, o comando “make clean”, pode ser utilizado para apagar os arquivos .o e gmon.out gerados