Trabalho Prático 2

O Analisador

Ítalo Dell'Areti

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

italodellareti@ufmg.br

1. Introdução.

O problema proposto é a criação de um contador de ocorrências das palavras usadas em um texto de acordo com uma nova ordem lexicográfica, sendo assim, podendo verificar a quantidade de vezes que uma determinada palavra aparece (ocorrência) e ordenar estes resultados de acordo com a ordem lexicográfica (ordem das letras do alfabeto organizadas da forma desejada). Portanto, a partir do input fornecido, contendo o arquivo de entrada com o texto a ser analisado e a ordem lexicográfica, vamos utilizar os tipos abstratos de dados, alocação dinâmica e processos de ordenação requeridos para resolver este problema. Tendo em vista isso, este documento possui como objetivo explicitar o funcionamento desta implementação.

2. Implementação

O programa foi desenvolvido utilizando a linguagem de programação C++, compilada pelo G++ da GNU Compiler Collection. Ademais, o programa foi produzido e testado em um ambiente Linux distribuição Ubuntu 20.04 LTS utilizando Visual Studio Code.

2.1 - Modularização e Organização

Temos como arquivos os headers alfabeto.h, io.h, lista.h, palavra.h, sort.h. A sort.h possui o header do algoritmo de ordenação Quicksort (requerido na especificação) e uma função que troca o valor de duas variáveis.

A implementação do programa foi realizada utilizando as classes **Alfabeto** e **Lista** e as estruturas **io**, **main** e **sort.** A **main** é responsável pela chamada de todas as funções, suas operações de escrita e leitura dos arquivos texto onde são armazenados os resultados, verificar as consistências de erro ao abrir arquivo de entrada, verifica se os dados da linha comando são possíveis e se as estruturas do input são válidas e se os tamanhos de mediana são viáveis.

A classe **Alfabeto** é responsável pela organização e estruturação da ordem lexicográfica, possuindo um array de tamanho 26, uma para cada letra do alfabeto.

Além disso, possui uma letra índice e cmp que realiza a comparação com as strings (Bem semelhante a função strcmp do c)

A classe **Lista** foi implementada devido a ser mais fácil ler os dados do texto em formato desta estrutura e depois é passado para um array, porque o acesso a lista seria ineficiente para o quicksort. Deste modo, a lista tem um nó para o primeiro elemento e para o último e tamanho, podendo inserir, retirar e encontrar qualquer palavra.

2.2 - Estrutura de Dados

Para atender as demandas em relação às operações necessárias, vamos analisar as principais funções do programa.

Em **alfabeto** temos as funções:

Alfabeto::Alfabeto : função para lidar com a ordem do alfabeto da forma mais simples e eficiente possível, um array de 26 letras indexado pela diferença das letras com 'A' assim A = 0, Z = 25 …

Então, ela é eficiente por comparar as letras A, B com a ordem customizada, só acessar o array no índice de A e no de B e comparar esses valores, se a ordem era inversa A = 25, B = 24, então B virá antes.

Alfabeto::indice : Pega o índice se for letra e se for qlq outra coisa, só retorna o valor na tabela ascii menos o 'A' pra ficar no mesmo deslocamento das letras a comparação é O(n), mas infelizmente não tem como fazer melhor

Alfabeto::cmp: Compara duas strings caractere por caractere.Possui a mesma ideia da função strcmp() do C ,negativo para uma string menor, 0 pra igual e positivo para maior.

Em **io** temos as funções:

dados_linha_comando: Esta Função é responsável pela leitura das opções que estão disponíveis quando se vai executar o programa, sendo elas -i (Arquivo de entrada para ser processado), -o (Endereço do arquivo de saída), -m (Tamanho da mediana de M elementos) e -s (Tamanho da partição)

parse_token : Verifica as posições da lista,verifica algumas condições e retorna o token que será utilizado.

eh_delimitador : Percorre os delimitadores e quando encontrar o caractere retorna verdadeiro.

limpa_texto : É utilizada para implementação do ponto extra. Para remover os caracteres indesejados do meio das palavras basta tirar todos os caracteres pois não fará diferença pois se eles estavam no meio de uma palavra, eles iriam “limpar” ela e não fariam nenhuma diferença fora de uma palavra. Assim, podemos remover

também porque mesmo que tenha um erro de digitação, como uma pontuação sem espaço, não dá para diferenciar de uma palavra que tem uma vírgula no meio, por exemplo.

Por conseguinte, o comportamento exato para isso basta converter o arquivo em uma string stream, passando de caractere a caractere, ignorando todo caractere indesejado.

Em **lista** temos as funções:

Lista::Lista : Cria a lista que vai ser utilizada para armazenar as palavras. Começa a lista com o primeiro e últimos elementos como NULL e o tamanho igual a 0.

Lista::insere : Insere novos elementos na lista.

Lista::atualiza : Procura a chave(palavra) na lista e insere naquela posição.

Lista::encontra : Percorre a lista até encontrar a palavra e conta suas ocorrências .

Lista::to_array : Função responsável por passar uma lista para array.

Lista::get_tamanho : Retorna o tamanho da lista.

Em **sort** temos as funções:

insertion_sort : Insertion sort padrão com ordenação decrescente de ordem $O(n^2)$, sendo o pior caso quando a entrada está ordenada em ordem decrescente com custo $O(n^2)$ e o melhor caso quando o array já está ordenado com custo O(n).

escolhe_pivot : Seleciona o pivo que será utilizado pelo Quicksort.

partition : Ele escolhe um elemento como pivô e particiona o array dado ao redor do pivô escolhido.

quicksort : Quicksort básico recursivo ,com ordenação de partições usando insertion sort. Se for menor que o parâmetro passado pela linha de comando -s E quicksort normal caso contrário. O pivô é escolhido a partir da mediana dos m primeiros elementos do vetor ordenados pelo insertion sort, se a partição for menor que o tamanho da mediana escolhida na linha de comando então só retorna o pivô da esquerda.

3. Análise Complexidade

3.1 - Tempo

Após esta breve visão sobre o programa como um todo, agora vamos mostrar algumas ordens de complexidade das principais funções e classes do programa, tendo em vista que seus impactos influenciam no tempo de execução do programa. Dessa forma, assim, a complexidade das principais funções são

Em **alfabeto** temos as funções:

Alfabeto::Alfabeto : Custo O(1)

Alfabeto::indice : Custo O(n)

Alfabeto::cmp: Custo O(n), pois compara duas strings de tamanho n.

Em **io** temos as funções:

dados_linha_comando: Custo O(1)

parse_token : Custo O(n)

eh_delimitador : Custo O(n)

limpa_texto : Custo O(n)

Em **lista** temos as funções:

Lista::insere : Custo O(1)

Lista::atualiza : Custo O(n)

Lista::encontra : Custo O(n)

Lista::to_array : Custo O(n)

Lista::get_tamanho : Custo O(n)

Em **sort** temos as funções:

insertion_sort : Custo $O(n^2)$

escolhe_pivot : Custo O(n)

partition : Custo O(n)

quicksort : Custo O(n log n)

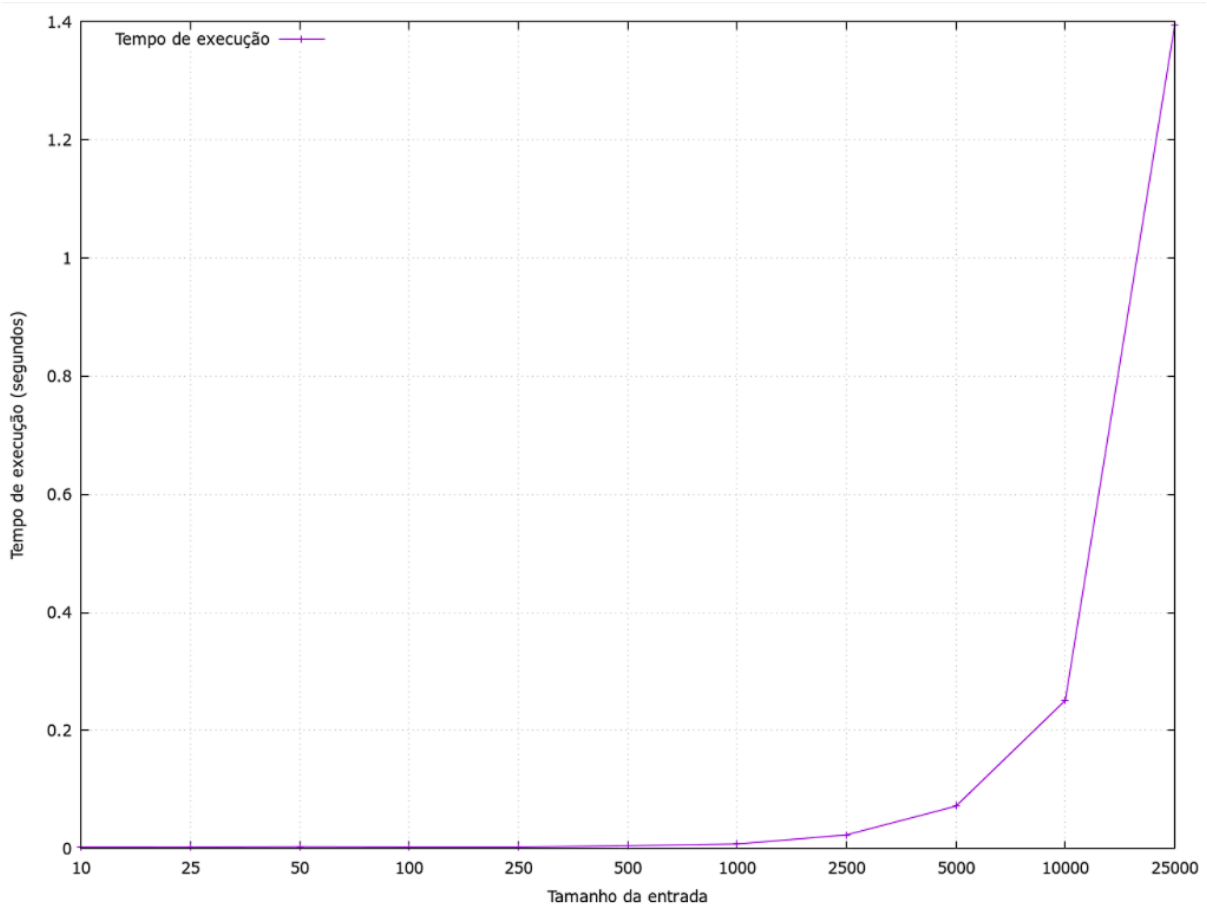
3.2 - Espaço

No pior dos casos, temos que considerar o tamanho das strings O(n*m), sendo n o número de palavras e m o número de caracteres da palavra, praticamente ocupando $O(n)$, .O restante ocupa O(1), constante.

4. Análise Experimental

4.1 - Desempenho computacional

Utilizando os resultados de testes de tempo de todo o programa, incluindo as operações de leitura e escrita de arquivos e alocação de memória, com a ordenação de 25000 palavras distintas, obtivemos o seguinte gráfico no gnuplot.



O comportamento gráfico de acordo com as entradas reafirma que a complexidade do algoritmo quicksort é O(n log n).

4.2 - Depuração de desempenho

Apenas com grandes quantidades de palavras distintas (acima de 4 mil) o ***gprof*** começa a acumular tempo na análise, com o método **Lista::encontra()** tendo cerca de 1 décimo de segundo com tempo de execução.

Saída da análise do gprof com a ordem lexicográfica normal (ordem alfabética normal) do livro Memórias Póstuma de Brás Cubas, o qual possui 6174 palavras distintas:

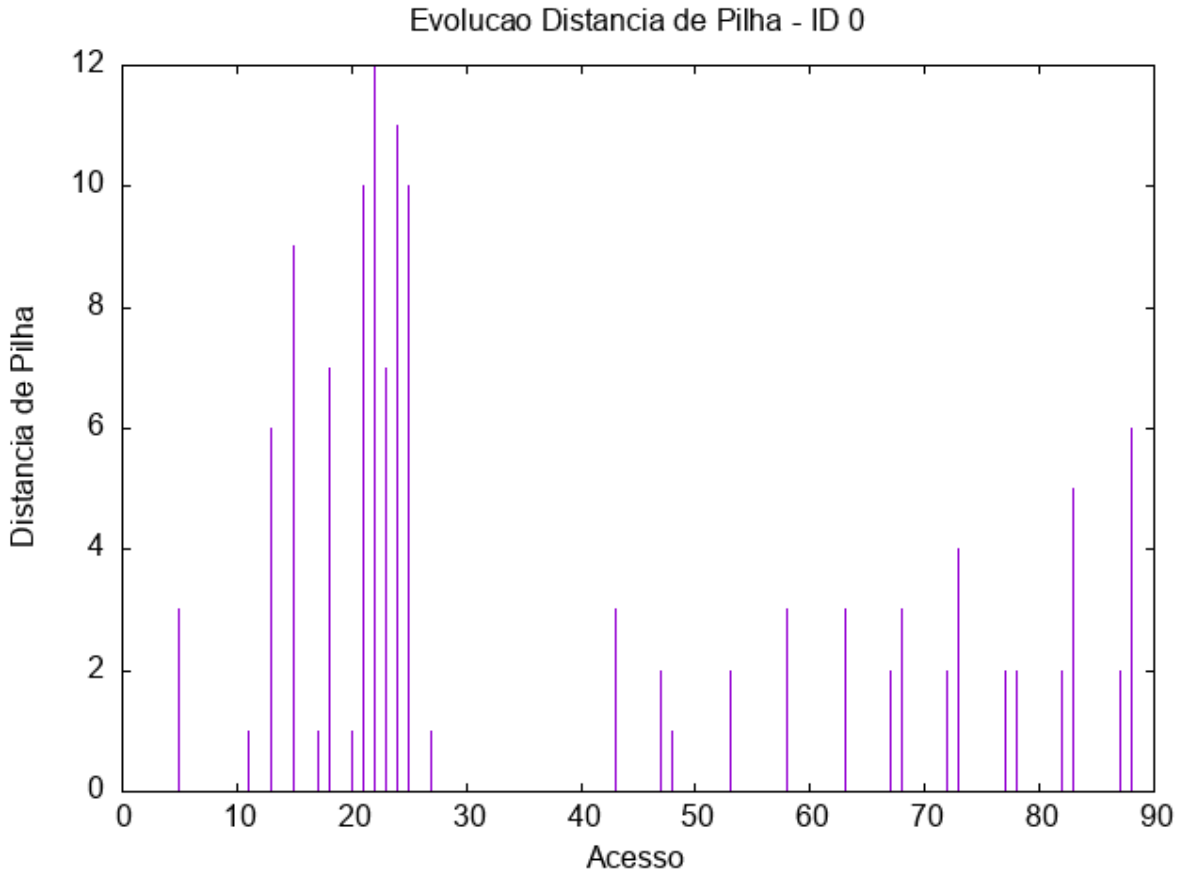
```
delareti@Delareti:~/analisador$ gprof bin/tp2 gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.12      0.13     0.13    26191     4.97     4.97  Lista::encontra(palavra_t const&)
  0.00      0.13     0.00   503332     0.00     0.00  Alfabeto::indice(char)
  0.00      0.13     0.00   100923     0.00     0.00  Alfabeto::cmp(palavra_t const&, palavra_t const&)
  0.00      0.13     0.00     6182     0.00     0.00  Lista::insere(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.13     0.00     1533     0.00     0.00  insertion_sort(palavra_t*, int, int, Alfabeto*)
  0.00      0.13     0.00        2     0.00     0.00  _init
  0.00      0.13     0.00        1     0.00     0.00  main
```

Os métodos com tempo igual à 0 foram cortados da imagem.

4.3 - Análise de Padrão de Acesso à Memória e Localidade de Referência

O teste foi realizado com a entrada de 25 mil palavras distintas e a realização de sua ordenação, para tal, a biblioteca ***memlog*** foi usada e os acessos de registro ao endereços foram salvos, utilizando o programa ***analisamem***, foram plotadas gráficos e histogramas sobre o acessos e ocupação de espaço.

Para a análise dos gráficos de distância de pilha e acesso,considere o ID 0 corresponde ao array do alfabeto e o ID 1 o array que é ordenado pelo Quicksort



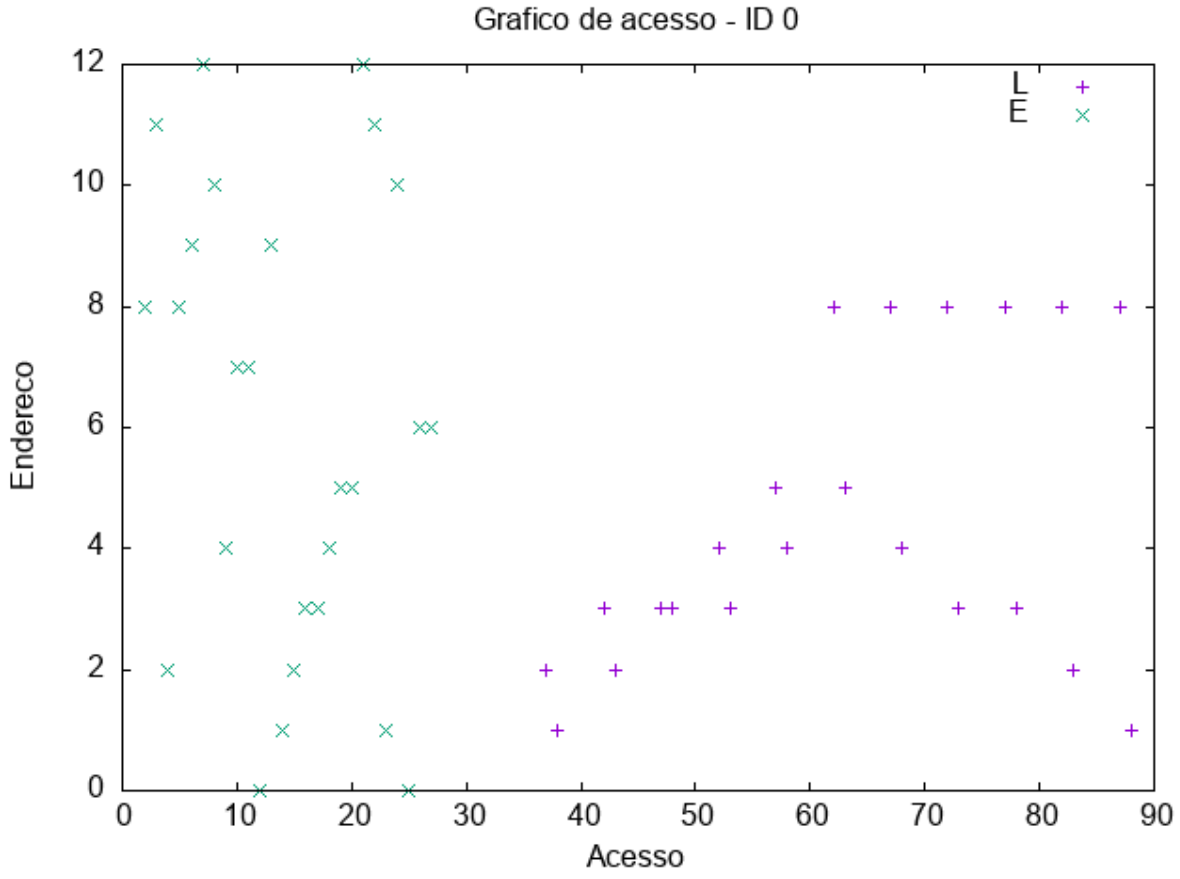
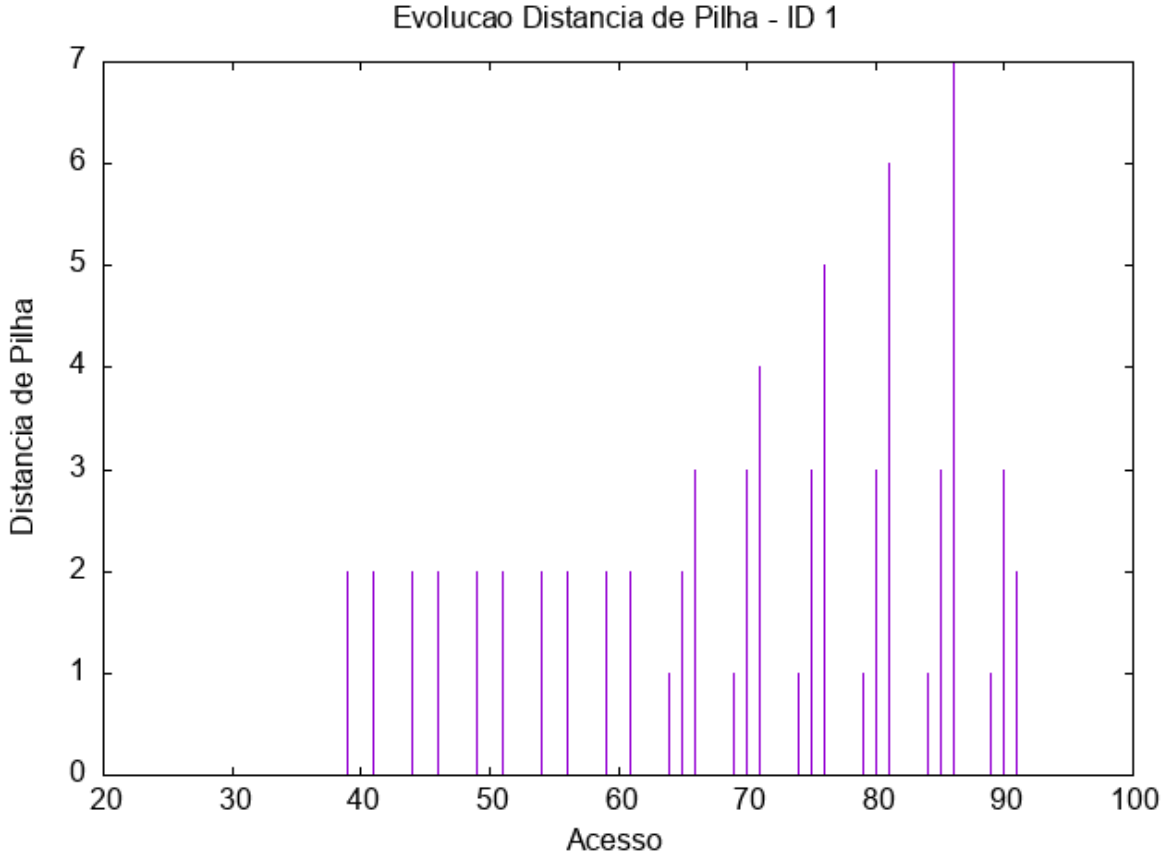
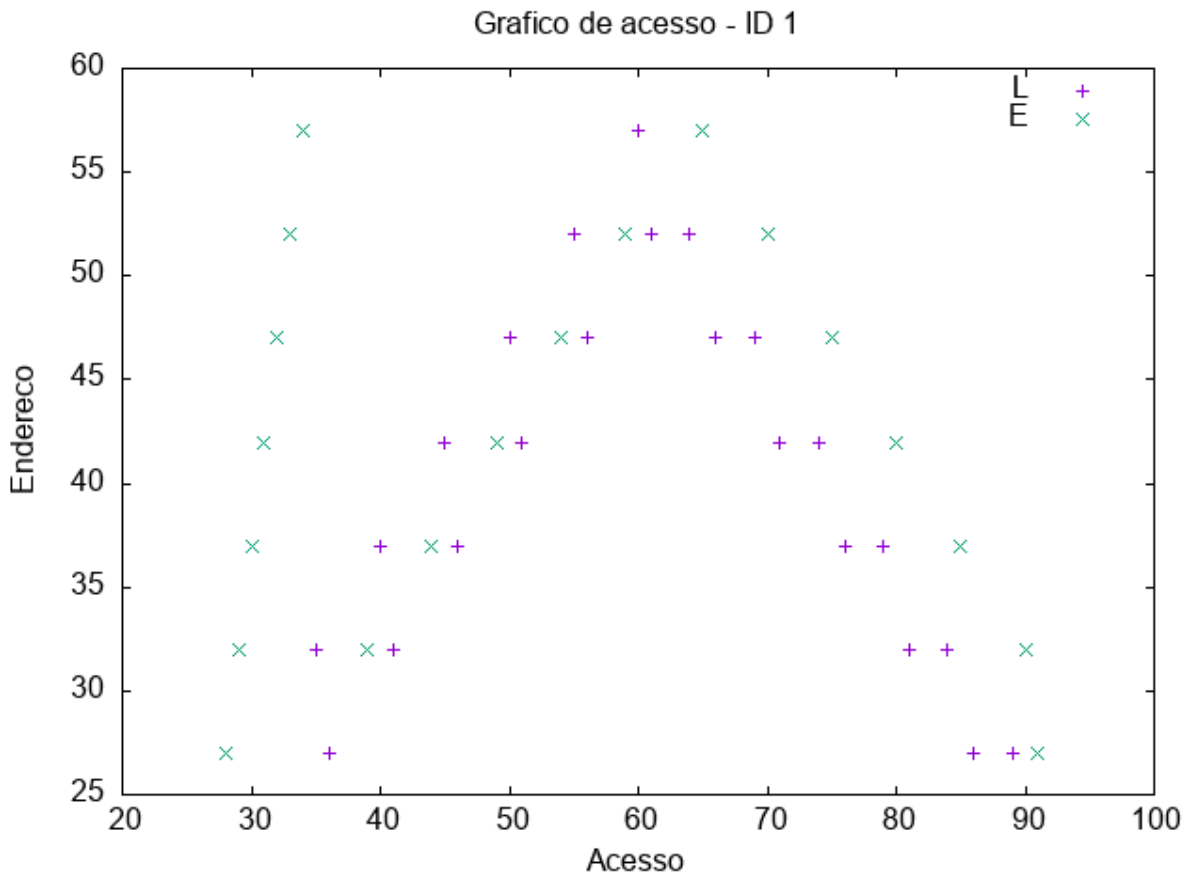


Grafico de acesso - ID 0
L
E
Endereco
Acesso
0
2
4
6
8
10
12
0
10
20
30
40
50
60
70
80
90


Evolucao Distancia de Pilha - ID 1
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
7
20
30
40
50
60
70
80
90
100



A distância de pilha do array alfabeto (ID 0) se inicia com uma grande frequência e ao passar do tempo vai diminuindo (fase 0) até a distância de pilha chegar a 0 (fase 2) devido a não ser mais utilizada pois foi passado os dados para o array que vai ordenar com o Quicksort.

Já em relação ao da ordenação (ID 1), a distância de pilha se inicia praticamente constante, pois está esperando os valores chegarem do alfabeto (fase 0)e após isso, crescendo regularmente(fase 2) e possuindo uma média de 2 e um total de tamanho 70.

Em relação ao acesso de memória dos ID 0 e 1, podemos resumidamente observar que no ID 0 o acesso inicia-se alto e com o passar do tempo diminui, ao contrário do ID 1 que se inicia baixo e com o passar do tempo aumente.Esse comportamento é devido a passagem de dados já explicada anteriormente.

Outrossim, o comportamento do gráfico de acesso possui as características esperadas de acordo com a estrutura de dados que são utilizados no alfabeto e ordenação com o quicksort.

5. Robustez

Algumas estratégias de robustez já foram citadas e explicadas durante a documentação, mas agora vamos ver algumas que ainda não foram citadas ou são de grande importância para o programa. Para o bom funcionamento do Quicksort os argumentos do tamanho da mediana e da partição são opcionais começam com

valores padrão para ignorar as modificações. Para lidar com argumentos inválidos o nome do arquivo deve ser informado, o tamanho da mediana não pode ser menor que 1 e o tamanho da partição não pode ser negativo.

Para lidar com a ausência do arquivo de entrada e de saída é obrigatório utilizá-los na hora de passar os parâmetros,pois se não vai receber um alerta e não poderá prosseguir. Na parte de case sensitive, onde ‘a’ == ‘ A’ é resolvida na função Alfabeto::Alfabeto, onde também é verificado se o alfabeto é inválido.

6. Testes

Para realizar os teste de funcionamento com as entradas fornecidas pelos professores

6.1 Caso de teste (2.tst)

```
#ORDEM
Q A Z W S X E D C R F V T G B Y H N M U J I K O L P
#TEXTO
Sofisticado, o Tenis Adidas Top Ten Low Sleek W e o tipo de calcado que faz uma baita diferenca no look da
Q
superfcies escorregadias. Mude os rumos de sua casualidade, va de Originals e faca efeito em
H
diversas ocasies.
```

Saída obtida

```
delareti@Delareti:~/analisador$
delareti@Delareti:~/analisador$
delareti@Delareti:~/analisador$ cat saida.txt
q 1
que 1
adidas 1
w 1
sua 1
superfcies 1
sofisticado 1
sleek 1
e 2
escorregadias 1
efeito 1
em 1
da 1
de 3
diferenca 1
diversas 1
casualidade 1
calcado 1
rumos 1
faz 1
faca 1
va 1
ten 1
tenis 1
tipo 1
top 1
baita 1
h 1
no 1
mude 1
uma 1
o 2
os 1
ocasies 1
originals 1
low 1
look 1
#FIM
delareti@Delareti:~/analisador$
```

6.2 Caso de desempate (4.tst)

```
#ORDEM
Q W E R   T Y U   I O   P   A S D F G   H   J   K   L   Z   X   C   V   B   N   M
#TEXTO
DFLHGLDKJHDLKJHVLJKVHSDKFHSDLFKJHLKJSHDJHSDFKJGHDJKHKJHDSFLKJDFHLKJH
SDKJGASK
DSVBD
DSKJHSDLKGHDGH
DJHGDJN
LSUcJ
ZKJXDHGJKSDFHKDFBVSDKVBDVHBSDKGSH
DJFBKG
487HS
ADSHGADSHGVGVADSHGDHGDDD.
```

Saída obtida

```
delareti@Delareti:~/analisador$ ./bin/tp2 -i entrada.txt -o saida.txt -m 5 -s 10
]delareti@Delareti:~/analisador$cat saida.txt
487hs 1
adshgadshgvgvadshgdhgddd 1
sdkjgask 1
dskjhsdlkghdgh 1
dsvbd 1
dflhgldkjhdlkjhvljkvhsdkfhsdlfkjhlkjshdjhsdfkjghdjkhkjhdsflkjdfhlkjh 1
djfbkg 1
djhgdjn 1
lsucj 1
zkjxdhgjksdfhkdfbvsdkvbdvhbsdkgsh 1
#FIM
delareti@Delareti:~/analisador$
```

6.3 Caso de desempate (5.tst)

```
#ORDEM
Q W E R   T Y U   I O   P   A S D F G   H   J   K   L   Z   X   C   V   B   N   M
#TEXTO
guarda-chuva
guarda-napo
guarda-coisa
guard-costas
```

Saída obtida

```
delareti@Delareti:~/analisador$ ./bin/tp2 -i entrada.txt -o saida.txt -m 5 -s 10
delareti@Delareti:~/analisador$ cat saida.txt
guard-costas 1
guarda-coisa 1
guarda-chuva 1
guarda-napo 1
#FIM
delareti@Delareti:~/analisador$
```

6.4 Caso de desempate (7.tst)

```
#ORDEM

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

#TEXTO
```

```
Dos quatro nomes, o que mais surpreende E o de Thiago Heleno! que
chegou a ser apontado em 2011 pelo entAo treinador Luiz Felipe Scolari
como o um dos melhores zagueiros do Brasil? Titular absoluto do time
com FelipAo, o jogador perdeu espaCo nesta temporada em decorrEncia de
sucessivas contusOes: O Palmeiras; assim, optou por nAo renovar seu
vInculo. Os outros trEs atletas haviam chegado ao Palestra ItAlia ao
longo deste ano e tambEm tinham seus contratos em vias de expirar.

VIAS DE EXPIRAR!

vIAs de ExPiRar?
```

Saída obtida

```
delareti@Delareti:~/trabalhos/analisador$ cat saida.txt
2011 1
que 2
quatro 1
e 2
espaco 1
expirar 3
entao 1
em 3
renovar 1
temporada 1
treinador 1
tres 1
titular 1
tinham 1
time 1
thiago 1
tambem 1
um 1
italia 1
o 5
outros 1
optou 1
os 1
perdeu 1
pelo 1
por 1
palestra 1
palmeiras 1
luiz 1
longo 1
jogador 1
heleno 1
haviam 1
felipe 1
felipao 1
de 5
deste 1
decorrencia 1
do 2
dos 2
ser 1
seu 1
seus 1
surpreende 1
sucessivas 1
scolari 1
a 1
atletas 1
ao 2
apontado 1
assim 1
absoluto 1
ano 1
zagueiros 1
contratos 1
contusoes 1
com 1
como 1
chegou 1
chegado 1
vias 3
vinculo 1
brasil 1
nesta 1
nomes 1
nao 1
melhores 1
mais 1
#FIM
```

Todas as saídas do programa produzido foram iguais aos inputs/ Outputs fornecidos.

Vídeo de alguns dos testes funcionando

Google Drive:
https://drive.google.com/file/d/19CFgWzvyknX4kh8uBN02JUegxo2q09MY/view?usp=sharing

Dropbox: https://www.dropbox.com/s/nr7vwv4dmyjhqbz/Analisador_final.mp4?dl=0

7. Conclusão

Ao final deste trabalho, podemos inferir que este possui o propósito de abordar e trabalhar e ampliar a visão sobre o funcionamento de ordenação de objetos e como elas se relacionam em uma aplicação, neste caso, de ordenação com Quicksort e algumas modificações de seus parâmetros com o seu pivô, sua mediana e o tamanho da sua partição.

Além disso, com as mudanças da organização das palavras, de acordo com cada organização lexicográfica amplia a complexidade de resolução do problema. Ademais, a visualização que poderíamos utilizar mais de uma estrutura de dados para melhorar o desempenho, como foi utilizado um lista simples e um array e ambos foram utilizados onde possuem melhor performance, passando da leitura do arquivo para lista e de lista para array para se utilizar melhor no Quicksort. Dessa forma, podendo praticar as mais diversas operações e modificações dos tipos de ordenações .

Bibliografia

BACKES, A. R. (2018). *Linguagem C - Completa e Descomplicada*

Geeks for geeks - Quicksort

Geeks for geeks - Insertion Sort

GNUPLOT - A Brief Manual and Tutorial

GPROF Tutorial – How to use Linux GNU GCC Profiling Tool

Instruções para compilação e execução

Para executar o programa basta dar um comando **make** para executar tudo. Alguns comandos adicionais são.

make all - compila tudo
make distclean - remove objetos apenas
make clean - remove objetos e executável

Após executar o make, basta executar este comando na pasta raiz

Exemplo: ./bin/tp2 -i entrada.txt -o saida.txt -m 5 -s 10

Sendo -i (Arquivo de entrada para ser processado)
-o (Endereço do arquivo de saída)
-m (Tamanho da mediana de M elementos)
-s (Tamanho da partição)

Passando o executável que está na pasta /bin e os parâmetros de arquivo de entrada, arquivo de saída, tamanho mediana e tamanho partição. Lembrando que para o normal funcionamento você deverá colocar valores válidos ou o programa irá retornar um aviso.

Sempre que mudar quaisquer valores de entrada, você deverá executar novamente o arquivo da pasta bin com seus respectivos parâmetros para atualizar os valores da saída para os valores atuais.

Não remover o arquivo log_mem.txt pois este foi utilizado para implementar a biblioteca **memlog** para estimar a performance do programa.