Trabalho Prático 2

Luiz Felipe de Sousa Faria – 2020027148

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lutilipe@ufmg.br

1. Introdução

O problema proposto foi implementar um contador do número de ocorrência de todas palavras em um dado texto e informar isso na saída do programa, em uma ordem lexicográfica previamente fornecida na entrada. Para a ordenação da saída, foi utilizado o algoritmo de QuickSort. Diante disso, o objetivo da presente documentação é analisar o desempenho, a robustez, a localidade de referência e distância de pilha das principais do programa, bem como refletir sobre algumas decisões e estruturas implementadas no projeto.

2. Método

2.1. Descrição

O programa principal se encontra no diretório ***TP***. Esse diretório é dividido em:

- ***include***: contém o arquivo *header* das classes *Word* (palavra), *AlphabeticOrder* (ordem alfabetica) e *Letter* (letra), do *memlog,* de funções de assert, de funções de utilidade (transformar uma *string* ou *char* para minúsculas) e de uma implementação de um Vetor;
- ***src***: contém a implementação das funções do include. A função *main* está em *analyzer.cpp*;
- ***obj***: contém os .o gerados pelo programa (vazio inicialmente)
- ***bin***: contém o binário do programa (vazio inicialmente)
- ***Makefile***: gera os arquivos binário e *.o* do programa e executa os testes de desempenho e análise de localidade.

O programa foi escrito em *C++* e compilado utilizando o *G++*.

O programa tem como entrada um arquivo que contém o texto na qual será realizada a contagem de palavras e uma nova ordem alfabética. Essa ordem alfabética define o valor dos caracteres de ***a-z***. O restante dos caracteres segue a tabela *ASCII*. O programa também recebe alguns parâmetros de entrada, que

definem o tamanho da partição do vetor para realizar o cálculo da mediana durante o QuickSort e o tamanho máximo do vetor para que o processo de ordenação seja feito por um algoritmo mais simples (insertion sort). O funcionamento da ordenação está descrito na seção 2.3.

Como saída, o programa gera o arquivo que contém todas as palavras do texto, seguido pela quantidade de vezes que elas aparecem. Essas palavras são ordenadas de acordo com a ordem passada no arquivo de entrada.

As principais classes do programa são:

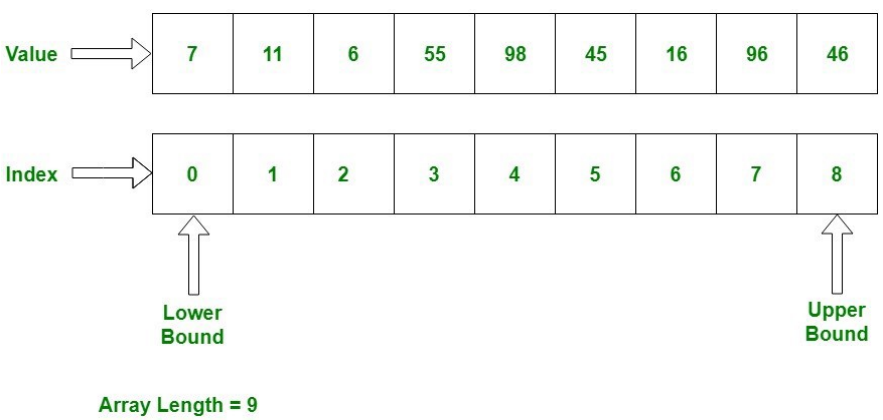
- ***Letter***: responsável por guardar as informações de uma letra (caracter da letra e valor na nova Ordem);
- ***Word***: armazena um vetor de letras (representando uma palavra) e a quantidade de vezes que essa palavra aparece no texto de entrada;
- ***WordVector***: vetor responsável por armazenar todas as palavras do texto, sem repetição, e realizar a ordenação delas;
- ***AlphabeticOrder***: armazena a nova ordem alfabética passada como parâmetro de entrada.

2.2. Estruturas de dados utilizadas

No projeto, a principal estrutura de dados utilizada foi o vetor. Os vetores (figura 1) são estruturas que armazenam sequencialmente na memória os dados. Eles podem ser estáticos, na qual seu tamanho é definido na compilação, ou dinâmicos, na qual seu tamanho é definido na execução e o mesmo é armazenado na memória *Heap*. Eles foram utilizados para armazenar todas as palavras do texto, sem repetição.

A escolha dessa estrutura de dados se deve ao método de ordenação utilizado, o QuickSort. Esse método realiza operações ***in-place*** sobre uma determinada lista, ou seja, não há a necessidade de alocar mais espaço na memória para fazer a ordenação sobre a entrada (como ocorre em outros algoritmos, como o MergeSort), já que o algoritmo modifica diretamente a lista. No entanto, algoritmos in-place ainda requerem a utilização de memória para realizar operações internas, como troca de valores e o armazenamento de variáveis temporárias.

Figura 1: vetores



2.3. Ordenação

A ordenação das palavras no arquivo de saída segue a ordem descrita na entrada do programa. Para tal, cada caractere assume um determinado valor. Os caracteres de a-z seguem o valor apresentado na nova ordem, e o restante segue a tabela ASCII. Nesse método de ordenação, todas as palavras estão em letra minúscula, o que elimina os caracteres maiúsculos do processo de ordenação.

O principal método de ordenação utilizado foi o *Quicksort*. Esse algoritmo é extremamente rápido e eficiente para entradas de tamanho suficientemente grandes, mas que também performa bem para entradas menores.

Esse algoritmo baseia-se no paradigma de “dividir para conquistar” (ou divisão e conquista), na qual divide-se o problema em problemas menores (divisão) e tenta achar uma solução para eles (conquista), para depois combiná-los para encontrar a solução do problema principal (combinação). No caso do Quicksort, as etapas são as seguintes:

- Escolhe-se arbitrariamente um elemento da lista para ser o pivô;
- Rearranja-se a lista para que todos os elementos da lista à esquerda do número escolhido como pivô sejam menores do que ele, e a direita, maiores do que ele. Com isso, o pivô estará na sua posição correta e serão geradas duas partições desordenadas da lista inicial;
- Utiliza-se o processo de recursão (ou seja, chamar novamente o algoritmo do *quicksort*) sobre as duas listas geradas, repetindo-se o processo até que todas as sub-partições estejam ordenadas.

Tal definição representa o algoritmo base do quicksort. No presente trabalho, foram realizadas algumas melhorias no algoritmo, para que o mesmo tivesse uma performance melhor em alguns casos. Essas melhorias foram: mediana de N (na qual N é um valor recebido como parâmetro na entrada) e o uso do Insertion sort para partições menores. O tamanho dessa partição menor também é recebido como parâmetro na entrada do programa.

Para o processo da mediana de N, são selecionados os primeiros N elementos da lista. Esses elementos são separados em uma lista, ordenados pelo método de *Insertionsort* para obter a mediana dentre os números selecionados. Utilizamos o índice do valor da mediana para ser o pivô do algoritmo de *quicksort.* Depois disso, o algoritmo de *quicksort* segue seu fluxo normalmente.

A ideia por trás dessa mediana é diminuir a arbitrariedade da escolha do pivô e tentar balancear seu valor conforme os outros valores da lista.

Já para partições menores, o algoritmo de quicksort acaba realizando mais operações do que o necessário para ordenar a lista. Com isso, a outra melhoria utilizada é a ordenação de partições menores utilizando um algoritmo mais simples (o escolhido foi o insertion sort), já que ele acaba sendo mais rápido para listas menores.

3. Análise de Complexidade e Espaço

A complexidade de tempo e espaço do programa reflete àquelas referentes ao algoritmo de Quicksort. Considerando **N** o tamanho de palavras do texto, **M** o número de elementos que serão utilizados para o cálculo da mediana em cada etapa e **S** o tamanho máximo da partição para o uso do *Insertion sort.*

O quicksort executa a ordenação em tempo O(N.log(N)). No entanto, com a existência de parâmetros não constantes M e S para controle da mediana e do tamanho máximo para utilizar outro método de ordenação mais simples, essa complexidade temporal acaba variando, tornando-se:

$$\mathbf{O((N-S).M^2.log(N-S)) + O(S^2) = O(max((N-S).M^2.log(N-S), S^2))}$$

Já para a complexidade de espaço, existem duas situações. Caso M seja maior que o tamanho inicial do vetor, a complexidade de espaço é **O(n)**, já que seria apenas para guardar o vetor contendo todas as palavras do texto, sem a utilização de variáveis temporárias que dependam da entrada. No entanto, caso M seja menor, haverá um espaço **O(N(N-M)**, pois para cada vez que executar a operação para obter a mediana da partição, é gerado um vetor temporário.

4. Estratégias de Robustez

Para garantir a robustez do programa, foram implementadas algumas estratégias para impedir valores indevidos de variáveis e acessos incorretos na memória.

Para o parâmetro que define o tamanho do sub-vetor da qual será obtido a mediana durante o quicksort, é checado se ele é menor ou igual ao tamanho da partição. Caso for maior, utiliza a posição central da partição como pivô. Caso não, segue o fluxo normal para encontrar o pivô a partir da mediana.

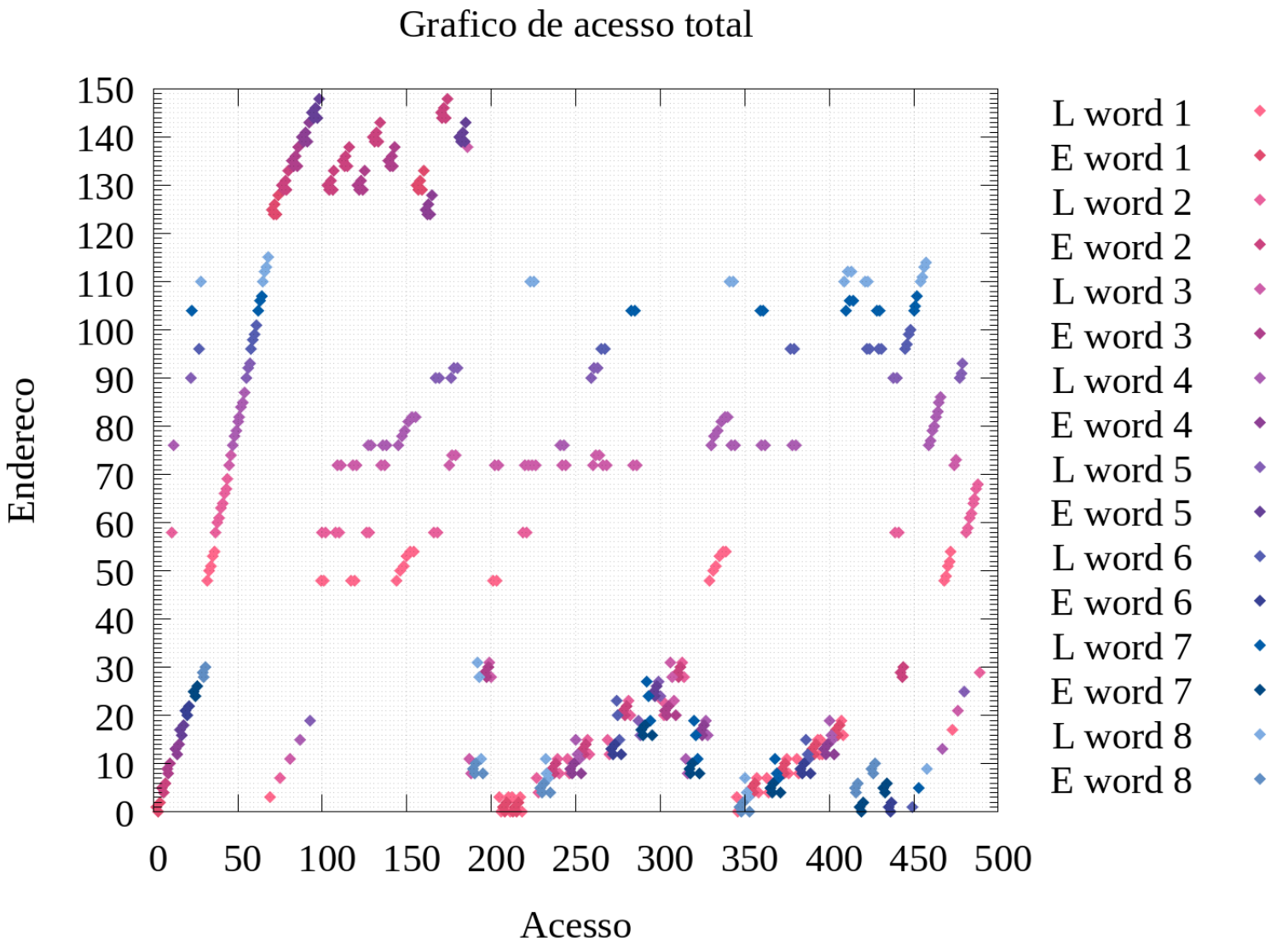
5. Análise Experimental

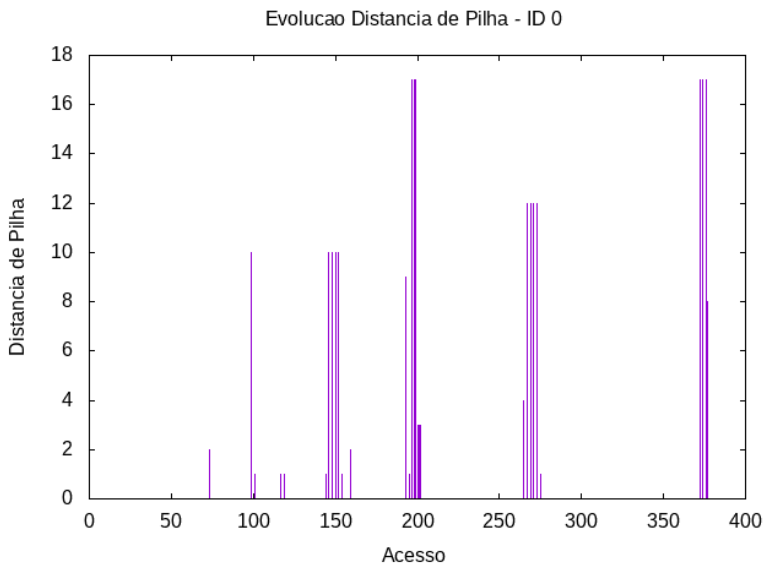
Para a análise experimental, foram realizados alguns experimentos a fim de medir a distância de pilha do programa, bem como essa distância de pilha varia com o tempo; calcular o tempo de execução das principais funções. Para tal, variou-se os seguintes parâmetros de entrada:

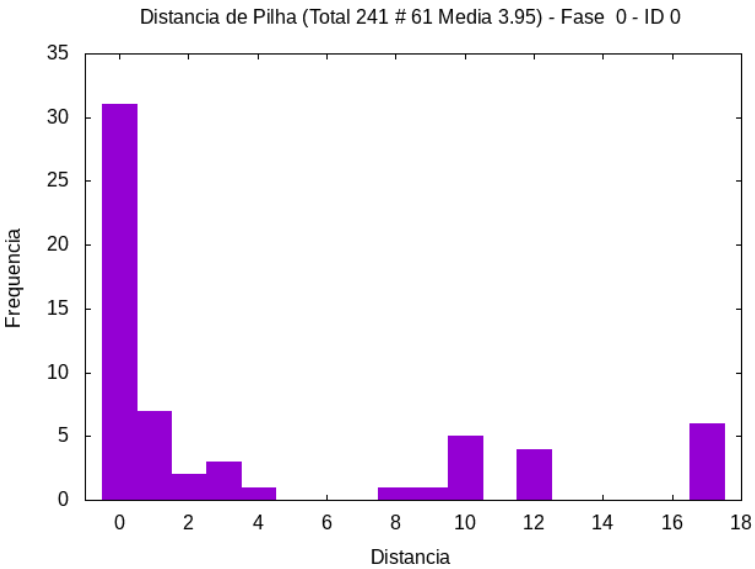
- Tamanho do vetor a ser ordenado
- Número de elementos considerados na escolha do pivô como mediana
- Tamanho da partição a partir do qual será usado um método simples.

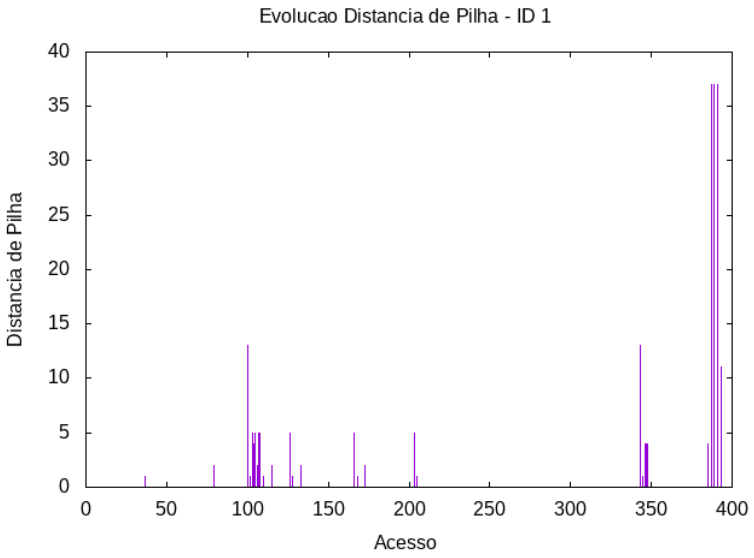
Tais parâmetros e seus respectivos valores são descritos em cada experimento realizado.

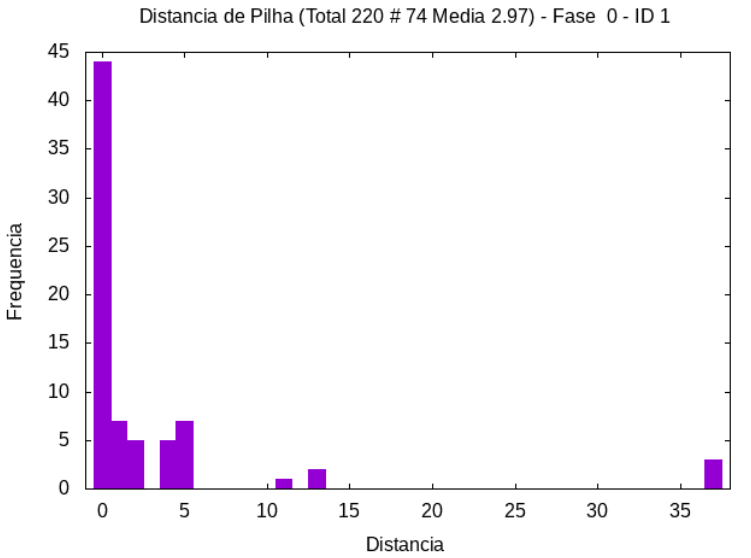
5.1. Localidade de referência e acesso à memória

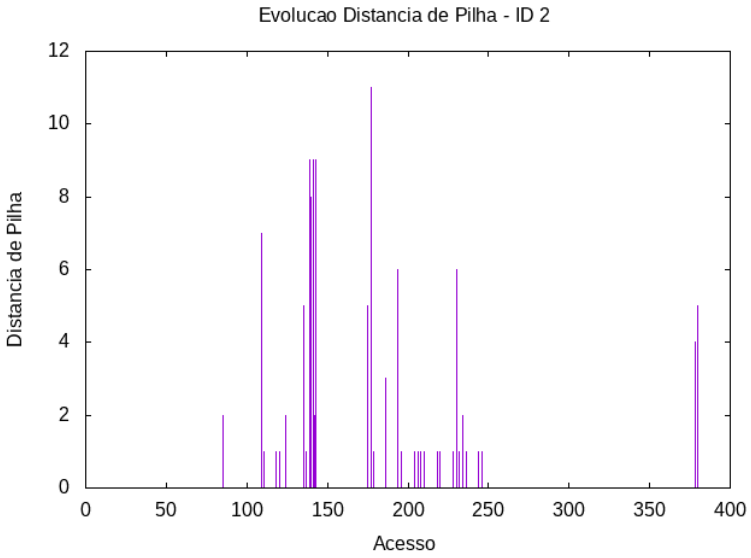


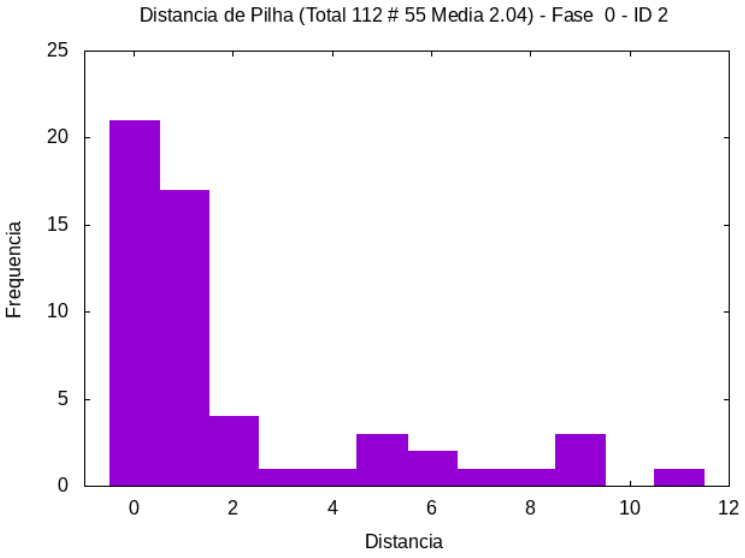


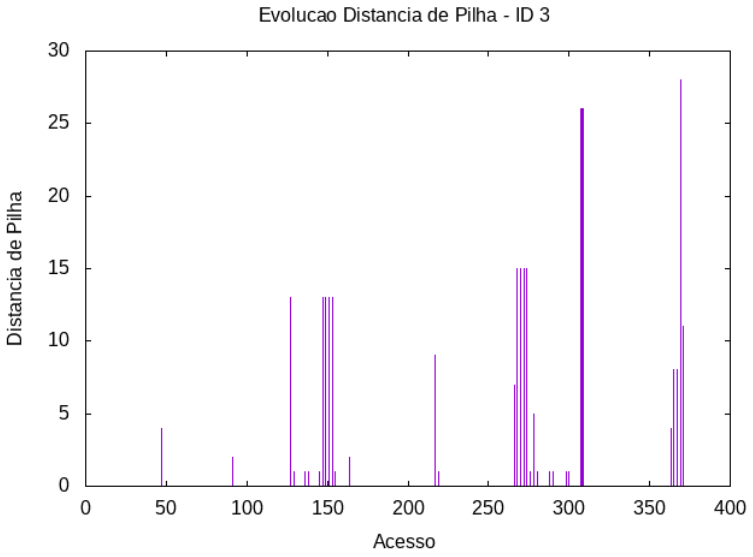


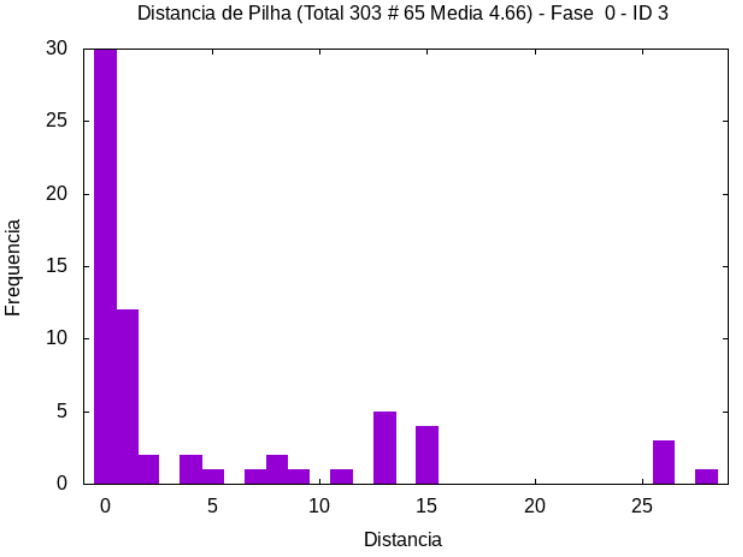

Evolucao Distancia de Pilha - ID 1
Distancia de Pilha
Acesso
0
5
10
15
20
25
30
35
40
0
50
100
150
200
250
300
350
400

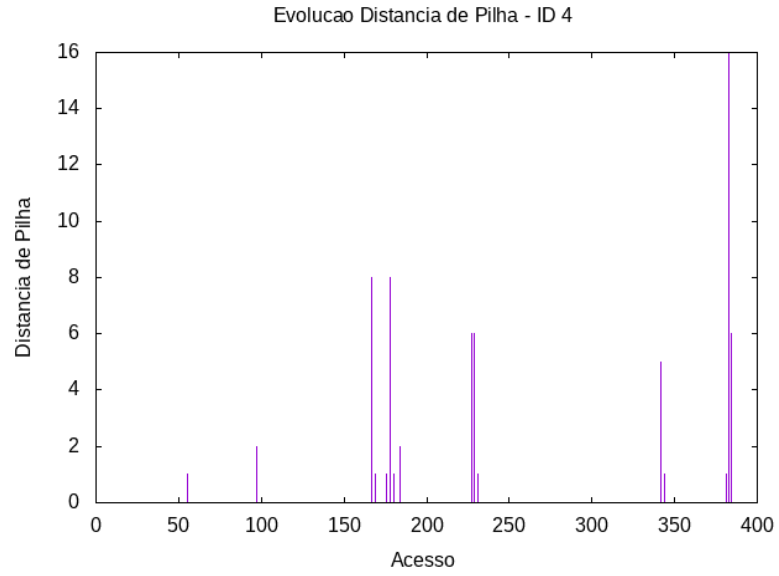

Distancia de Pilha (Total 220 # 74 Media 2.97) - Fase 0 - ID 1
Frequencia
Distancia
0
5
10
15
20
25
30
35
40
45
0
5
10
15
20
25
30
35

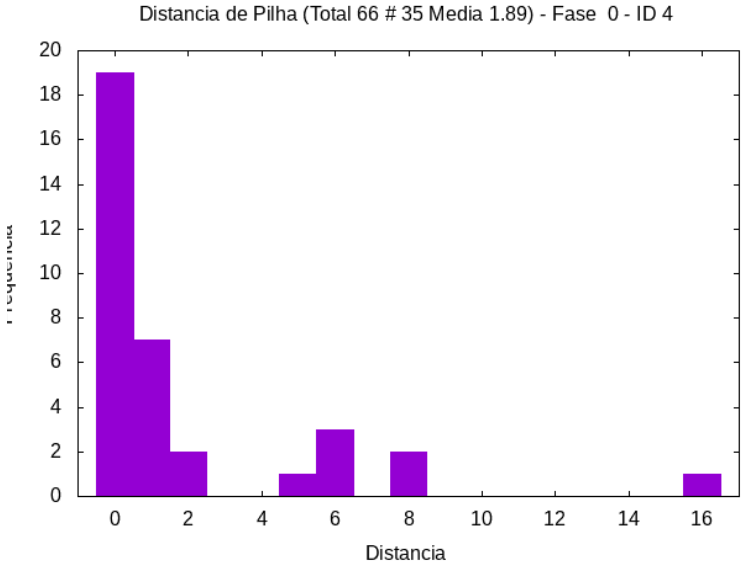

Evolucao Distancia de Pilha - ID 2
Distancia de Pilha
Acesso
0
2
4
6
8
10
12
0
50
100
150
200
250
300
350
400

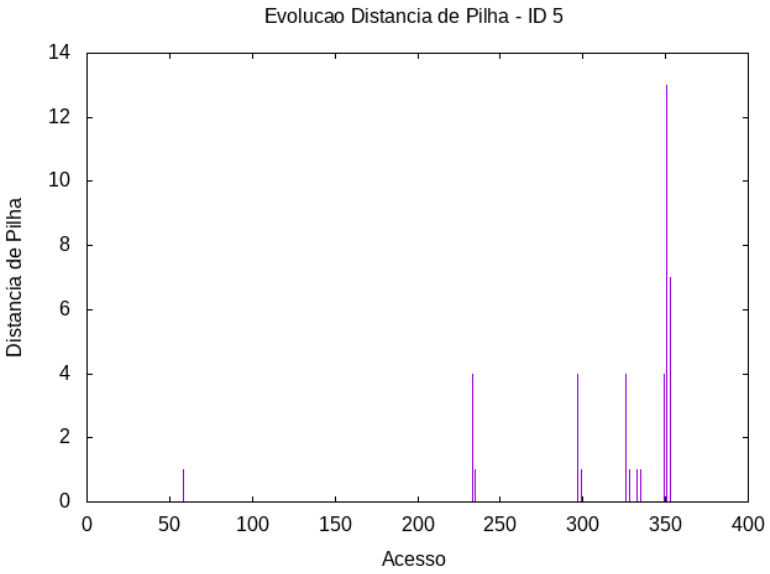

Distancia de Pilha (Total 112 # 55 Media 2.04) - Fase 0 - ID 2
Frequencia
Distancia
0
5
10
15
20
25
0
2
4
6
8
10
12

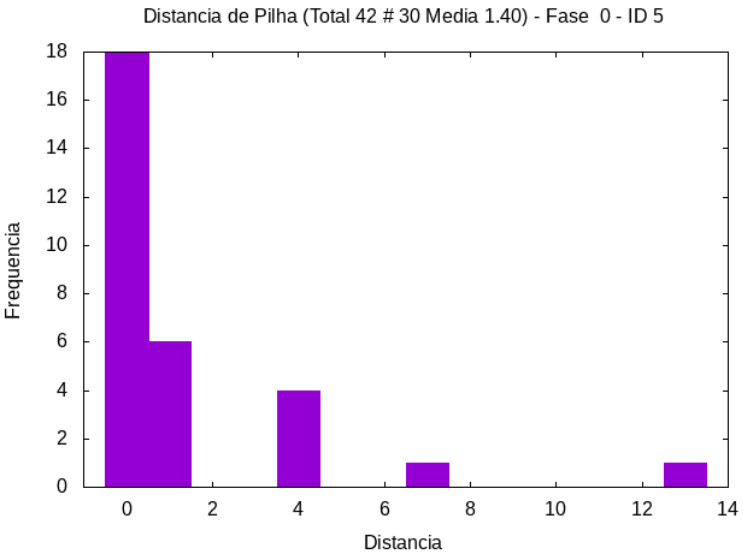

Evolucao Distancia de Pilha - ID 3
Distancia de Pilha
Acesso
0
5
10
15
20
25
30
0
50
100
150
200
250
300
350
400

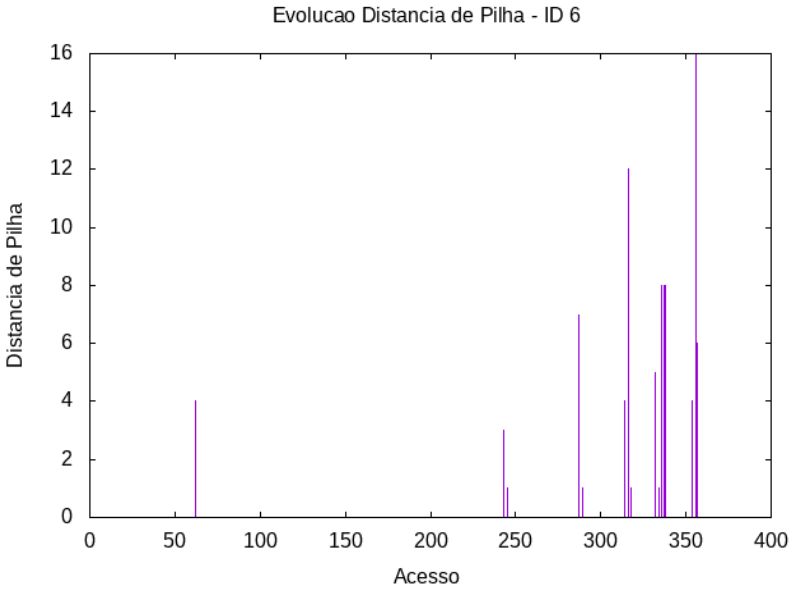

Distancia de Pilha (Total 303 # 65 Media 4.66) - Fase 0 - ID 3
Frequencia
Distancia
0
5
10
15
20
25
30
0
5
10
15
20
25

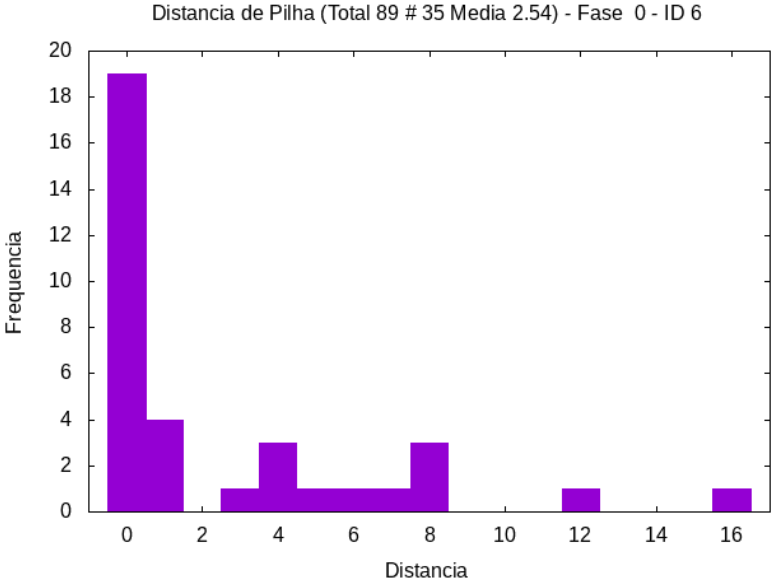

Evolucao Distancia de Pilha - ID 4
Distancia de Pilha
Acesso
0
2
4
6
8
10
12
14
16
0
50
100
150
200
250
300
350
400

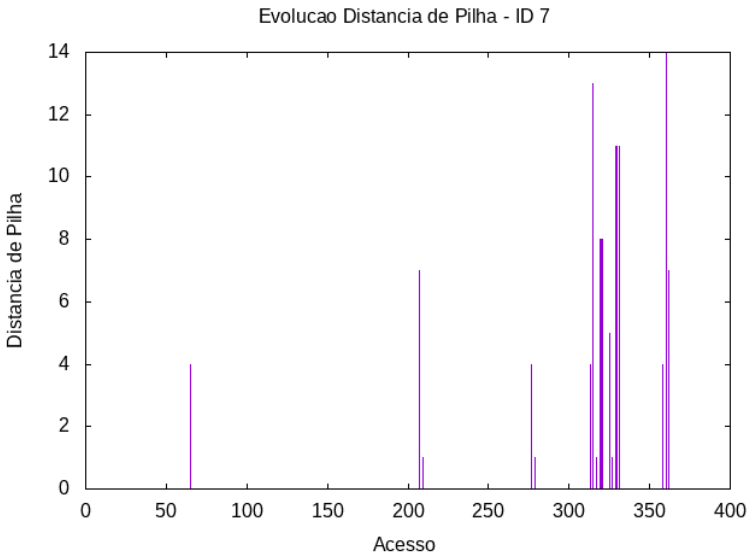

Distancia de Pilha (Total 66 # 35 Media 1.89) - Fase 0 - ID 4
Distancia
0
2
4
6
8
10
12
14
16
18
20

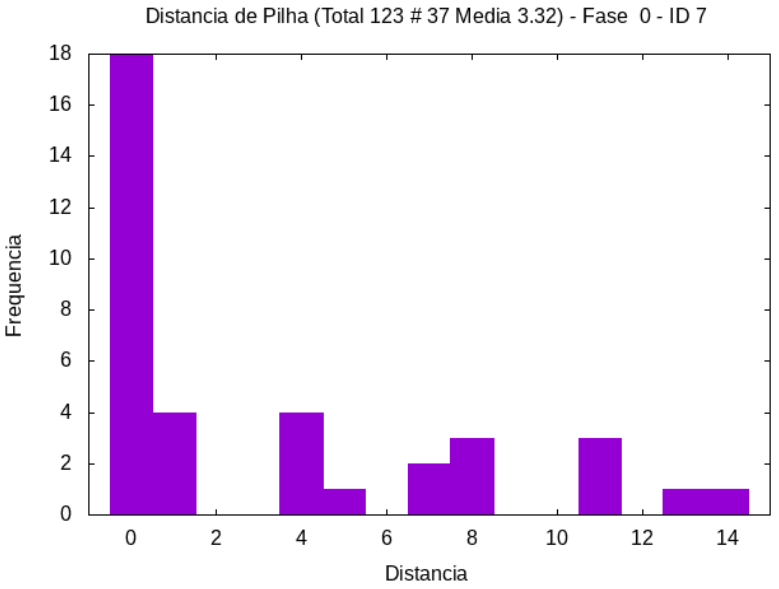

Evolucao Distancia de Pilha - ID 5
Distancia de Pilha
Acesso
0
2
4
6
8
10
12
14
0
50
100
150
200
250
300
350
400


Distancia de Pilha (Total 42 # 30 Media 1.40) - Fase 0 - ID 5
Frequencia
Distancia
0
2
4
6
8
10
12
14
16
18


Evolucao Distancia de Pilha - ID 6
Distancia de Pilha
Acesso
0
2
4
6
8
10
12
14
16
0
50
100
150
200
250
300
350
400


Distancia de Pilha (Total 89 # 35 Media 2.54) - Fase 0 - ID 6
Frequencia
Distancia
0
2
4
6
8
10
12
14
16
18
20





Para o cálculo da localidade de referência e criação do mapa de acesso à memória, considerou-se uma entrada contendo 8 palavras, utilizando uma mediana de 5 e o tamanho máximo para uso do InsertionSort como 3.

Cada palavra no mapa é representada por uma cor de leitura e uma cor de escrita (**L** e **E**, respectivamente).

As distâncias de pilha revelam que há uma grande quantidade de acessos com distâncias pequenas. Isso se deve, principalmente, aos acessos durante a execução do Quicksort, na qual são realizadas diversas operações de comparação e troca de valores *in-place* no vetor. Esse processo pode ser visualizado no mapa na parte inferior, na qual há bastante acesso de leitura e escrita.

Percebe-se que, apesar da quantidade de palavras no texto ser baixa, a distância de pilha para cada uma é relativamente alta. Isso se deve às diversas comparações que são realizadas no processo de ordenação. E, à medida que os parâmetros de entrada são modificados, o número tende a flutuar. Por exemplo, se aumentarmos o valor do parâmetro M, o número de comparações tenderia a diminuir, pois ele realizaria o cálculo da mediana para uma quantidade menor de partições e utilizaria o valor do meio para aquelas partições que possuem um tamanho menor do que M.

Encontrar um valor ideal para M e S impacta no número de acessos em memória, no tempo de execução do programa e na distância de pilha. Tal valor foi estimado na seção seguinte, através de alguns experimentos.

5.2. Desempenho computacional

1)

- 20 mil palavras únicas
- Mediana de 3
- Tamanho da partição 64 para executar o *Insertionsort*

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
69:51	0.50	0.50	199923038	0.00	0.00	Word::operator==(Word&)
22.24	1.06	0.16	1	0.00	0.00	handleInput()
4.17	1.09	0.03	53392618	0.00	0.00	Letter::getVal()
3.18	1.11	0.02	4903442	0.00	0.00	Letter::getIndex()
1.39	1.12	0.01	1	0.00	0.00	Word::adaptToNewAlphabeticOrder(AlphabeticOrder&)
0.00	1.12	0.00	882703	0.00	0.00	Word::operator=(Word&)
0.00	1.12	0.00	743069	0.00	0.03	Word::operator>(Word&)
0.00	1.12	0.00	157000	0.00	0.00	Letter::setIndex(int)
0.00	1.12	0.00	156974	0.00	0.00	AlphabeticOrder::getOrder()
0.00	1.12	0.00	20027	0.00	0.00	stringToLowerCase()
0.00	1.12	0.00	20001	0.00	0.00	removeUnexpectedChars()
0.00	1.12	0.00	20000	0.00	0.00	Word::Word()
0.00	1.12	0.00	19996	0.00	0.00	WordVector::push(Word)
0.00	1.12	0.00	19996	0.00	0.00	Word::getReps()
0.00	1.12	0.00	19996	0.00	0.00	Word::toString()
0.00	1.12	0.00	2901	0.00	0.00	Word::getId()
0.00	1.12	0.00	967	0.00	0.07	void insertionSort<TmpWord>(TmpWord*, int, int)
0.00	1.12	0.00	967	0.00	14.58	WordVector::partition(Word*, int, int)
0.00	1.12	0.00	30	0.00	0.00	charToLowerCase(char)
0.00	1.12	0.00	4	0.00	0.00	Word::increaseReps()
0.00	1.12	0.00	2	0.00	0.00	getInputNumber(char*)

2)

- 20 mil palavras únicas
- Mediana de 100
- Tamanho da partição 3 para executar o *Insertionsort*

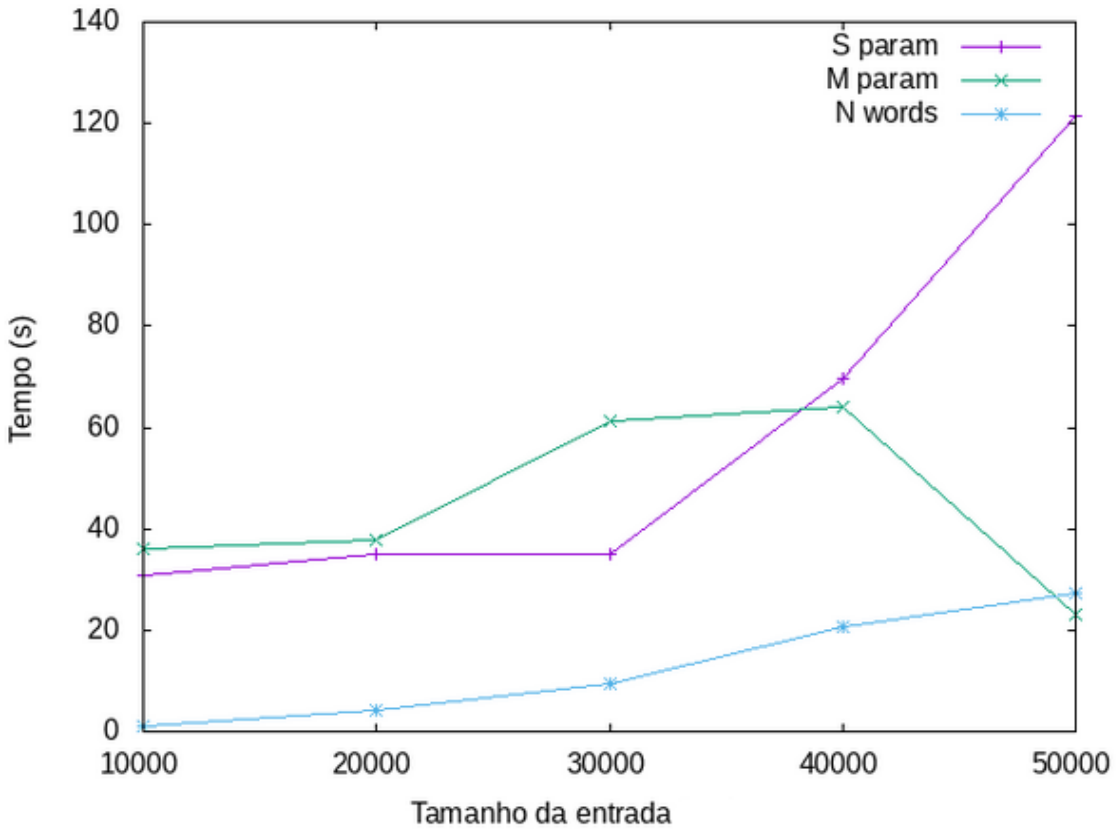
% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
63:05	0.49	0.49	199923038	0.00	0.00	Word::operator==(Word&)
22.10	1.06	0.17	1	0.00	0.00	handleInput()
7.15	1.11	0.06	1	0.00	0.00	Word::adaptToNewAlphabeticOrder(AlphabeticOrder&)
4.30	1.14	0.03	53392618	0.00	0.00	Letter::getVal()
4.30	1.17	0.03	177448	0.03	0.03	Word::operator>(Word&)
0.00	1.17	0.00	8753746	0.00	0.00	Letter::getIndex()
0.00	1.17	0.00	1301233	0.00	0.00	Word::operator=(Word&)
0.00	1.17	0.00	157000	0.00	0.00	Letter::setIndex(int)
0.00	1.17	0.00	56974	0.00	0.00	AlphabeticOrder::getOrder()
0.00	1.17	0.00	30200	0.00	0.00	Word::getId()
0.00	1.17	0.00	20027	0.00	0.00	stringToLowerCase()
0.00	1.17	0.00	20001	0.00	0.00	removeUnexpectedChars()
0.00	1.17	0.00	20000	0.00	0.00	Word::Word()
0.00	1.17	0.00	19996	0.00	0.00	WordVector::push(Word)
0.00	1.17	0.00	19996	0.00	0.00	Word::getReps()
0.00	1.17	0.00	19996	0.00	0.00	Word::toString[abi:cxx11]()
0.00	1.17	0.00	8114	0.00	4.08	WordVector::partition(Word*, int, int)
0.00	1.17	0.00	302	0.00	66:36	void insertionSort<TmpWord>(TmpWord*, int, int)
0.00	1.17	0.00	30	0.00	0.00	charToLowerCase(char)
0.00	1.17	0.00	4	0.00	0.00	Word::increaseReps()
0.00	1.17	0.00	2	0.00	0.00	getInputNumber(char*)

3)

- 40 mil palavras únicas
- Mediana de 1
- Tamanho da partição 3 para executar o *Insertionsort*

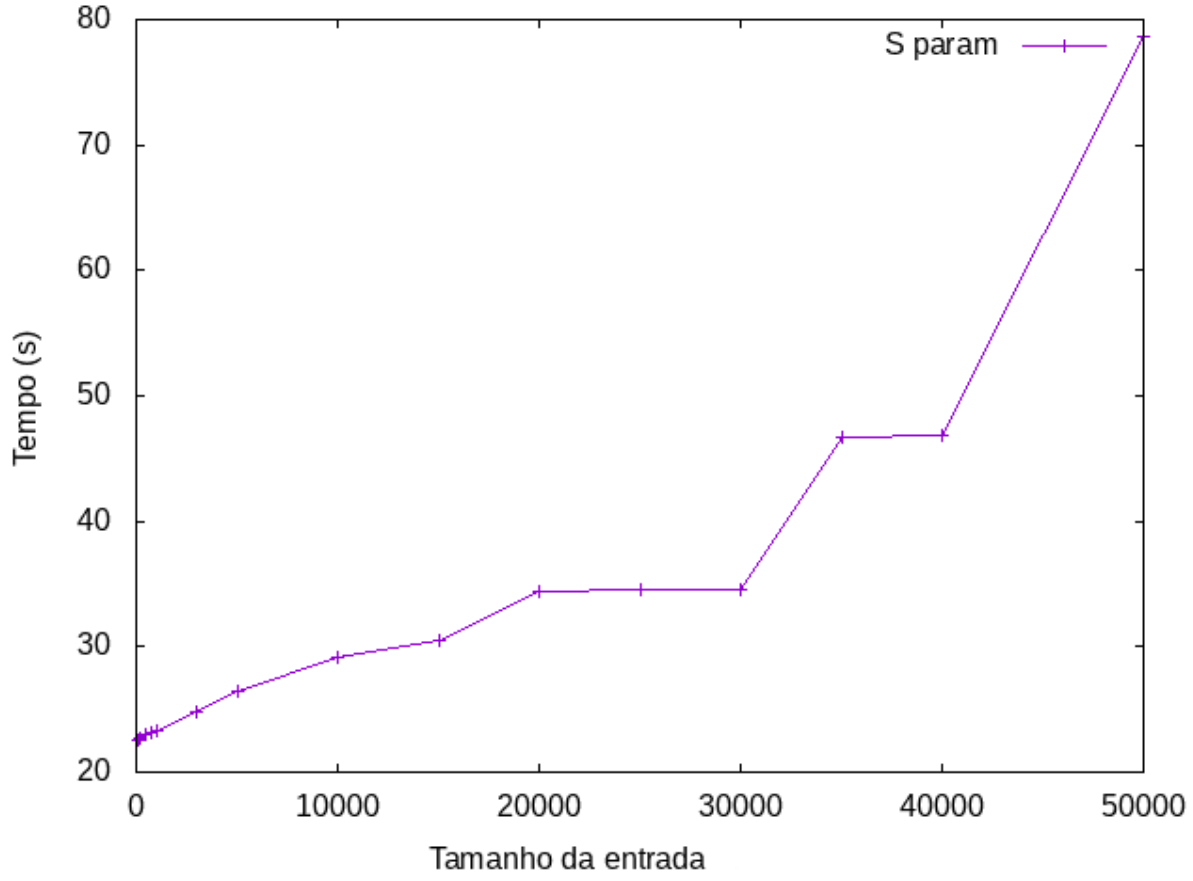
% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
48:37	3.03	3.03	210351538	0.01	0.01	Letter::getVal()
37:28	5.37	2.34	784053452	0.00	0.01	Word::operator==(Word&)
11.26	6.05	1.08	1	0.00	0.00	handleInput()
2.00	6.18	0.13	1	0.00	0.00	Word::adaptToNewAlphabeticOrder(AlphabeticOrder&)
1.12	6.25	0.07	11060728	0.01	0.01	Letter::getIndex()
0.32	6.27	0.02	1551230	0.01	0.06	Word::operator>(Word&)
0.16	6.28	0.01	4552	2.20	19.25	WordVector::partition(Word*, int, int)
0.00	6.28	0.00	2009371	0.00	0.00	Word::operator=(Word&)
0.00	6.28	0.00	329974	0.00	0.00	Letter::setIndex(int)
0.00	6.28	0.00	329948	0.00	0.00	AlphabeticOrder::getOrder()
0.00	6.28	0.00	40027	0.00	0.00	stringToLowerCase()
0.00	6.28	0.00	40001	0.00	0.00	removeUnexpectedChars()
0.00	6.28	0.00	40000	0.00	0.00	Word::Word()
0.00	6.28	0.00	39567	0.00	0.00	WordVector::push(Word)
0.00	6.28	0.00	39567	0.00	0.00	Word::getReps()
0.00	6.28	0.00	39567	0.00	0.12	Word::toString()
0.00	6.28	0.00	4552	0.00	0.00	void insertionSort<TmpWord>(TmpWord*, int, int)
0.00	6.28	0.00	4552	0.00	0.00	Word::getId()
0.00	6.28	0.00	433	0.00	0.00	Word::increaseReps()
0.00	6.28	0.00	30	0.00	0.00	charToLowerCase(char)
0.00	6.28	0.00	2	0.00	0.00	getInputNumber(char*)

Para o primeiro gŕafico, variou-se a quantidade de palavras de 10 mil a 50 mil (linha azul), e depois fixou-se esse número em 50 mil e variou o parâmetro M (linha verde) e S (linha roxa) de 1000 a 5000. Observa-se que se S for igual a N, o algoritmo de Quicksort não ocorre. Se M = N, faz-se a escolha do pivô utilizando a mediana de M apenas uma vez. O restante, é feito como mediana de 1.



Para o segundo gráfico, buscou-se entender o comportamento do parâmetro S, a fim de encontrar uma solução ideal. Para tal, considerou-se a mesma entrada de 50 mil palavras únicas e a mediana de 3.

Nele, é possível perceber que a medida que aumentamos S, a função vai se tornando quadrática (conforme a complexidade assintótica descrita na seção 3). Até aproximadamente o valor 1000 do parâmetro S, a função se mantém em um platô que é de 22s. À medida que vamos aumentando isso, a função começa a crescer de forma semelhante à uma função quadrática. Isso leva a crer que o valor máximo para o parâmetro S e para uma entrada de 50 mil palavras, utilizando a mediana de 3, é de 1000. Para outros parâmetros (principalmente variando M), outros valores de S serão encontrados.



6. Conclusão

O presente trabalho propôs-se a implementar e descrever a contagem de palavras em um texto em C++, utilizando vetores e os algoritmos de *Quicksort* e *InsertionSort*, bem como fazer a análise computacional de desempenho e acesso à memória das operações realizadas.

A análise feita sobre o programa acaba se tornando mais complexa, devido a complexidade dos algoritmos implementados. No entanto, com a geração de gráficos e tabelas feitas durante as análises, é possível visualizar o comportamento da estrutura com diferentes tamanhos de entradas (tamanho do texto, número

máximo para ordenar utilizando o InsertionSort e tamanho do vetor para o cálculo da mediana) ao longo da execução do programa, tornando a análise mais simples.

7. Bibliografia

CHORMEN, Thomasb H. *et al*. **Algoritmos - Teoria e Prática**. 3ª edição. Capítulo 3: Crescimento de funções. Editora Elsevier, 10 de abril de 2012.

CHORMEN, Thomasb H. *et al*. **Algoritmos - Teoria e Prática**. 3ª edição. Capítulo 7: *Quicksort*. Editora Elsevier, 10 de abril de 2012.

APÊNDICE A - INSTRUÇÕES PARA COMPILAÇÃO E EXECUÇÃO

A compilação do programa pode acontecer apenas rodando o comando **make**.

Para executá-lo, utilize o comando:

bin/tp2 -[i|I] <arq> **-[o|O]** <arq> **-[m|M]** <number> **-[s|S]** <number>

- **[i|I]:** representa o caminho para o arquivo contendo a entrada do programa.
- **[o|O]:** representa o caminho para o arquivo na qual será escrita a saída do programa.
- **[m|M]:** representa o número M da mediana para realizar as partições.
- **[s|S]:** representa o número máximo S para realizar a ordenação em uma partição utilizando o *InsertionSort*