Trabalho Prático 2
Análise Lexicográfica

Guilherme Mota Bromonschenkel Lima - 2019027571

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

guilhermebkel@ufmg.br

1. Introdução

O problema proposto foi implementar um programa que fizesse uma análise de ocorrências de palavras em um texto baseado em uma nova ordem lexicográfica - também chamada popularmente de ordem alfabética, que é extremamente útil para realizar a ordenação de palavras que usamos diariamente.

Ao executar o programa, ele faz a leitura de um arquivo contendo a nova ordem lexicográfica, além do texto que será analisado.

É possível utilizar alguns argumentos durante a execução do programa para indicar qual será o arquivo de entrada utilizado, o caminho do arquivo de saída contendo o resultado, além de configurações que serão utilizadas no algoritmo de ordenação QuickSort durante a consolidação do resultado.

Ao fim da execução do programa, é criado um arquivo de saída que contém a quantidade de ocorrências de cada palavra do texto. Além disso, esse resultado é ordenado utilizando a nova ordem lexicográfica que foi fornecida na inicialização do programa.

Durante a execução do programa, fizemos uso de um módulo chamado **memlog** para fazer diversos registros de acesso de memória. Além disso, para realizar a análise experimental foi necessário utilizar a biblioteca **analisamem**, que é responsável por pegar os dados gerados pela biblioteca **memlog** e tratá-los para uma melhor visualização.

Por fim, fazemos uso também de uma ferramenta chamada **gprof** para verificar métricas de desempenho computacional, como a duração de chamadas de funções do programa.

2. Implementação

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

Os testes do programa foram realizados no sistema operacional Ubuntu (versão 18.04).

2.1. Estrutura de Dados

Por ser um programa mais simples em questão de quantidade de modelos, foi utilizado apenas um modelo principal chamado WordOccurence (guarda a quantidade de ocorrências de uma palavra, além de seu valor formatado e original) que foi criado através do uso de struct.

Sendo assim, como nessa análise lexicográfica precisamos ler diversas palavras de um texto, foi necessário fazer uso da uma estrutura de dados ArrangementList (é uma estrutura de lista adaptada para facilitar seu uso, que foi implementada manualmente) para armazenar esses dados em uma lista.

2.2. Formato dos dados

Durante a execução do programa é feita a leitura de um arquivo de entrada para obter os dados que serão utilizados mais tarde.

Nesse sentido, vale ressaltar que durante a leitura do arquivo de entrada nós buscamos os dados de acordo com as palavras reservadas “#ORDEM” e “#TEXTO”. Quando essas palavras aparecem no arquivo, significa que as próximas leituras são de valores específicos de cada contexto (como a nova ordem lexicográfica ou as palavras do texto).

Todas as letras utilizadas na nova ordem lexicográfica são formatadas para caixa baixa a fim de facilitar as comparações que são feitas posteriormente.

Além disso, as palavras do texto são salvas em seu formato original (para fazer a comparação da ordem lexicográfica) e também formatadas (removendo caracteres especiais e deixando em caixa baixa para serem mostradas no resultado da execução do programa).

2.3. Ordenação dos dados

Para realizar a ordenação dos dados durante o cálculo do resultado final da execução do programa nós utilizamos o algoritmo **QuickSort**.

Vale ressaltar que estendemos esse algoritmo para que, através de alguns argumentos de inicialização do programa, ele utilize um algoritmo de ordenação mais simples, que no nosso caso é o **SelectionSort**, para as partições que tenham um tamanho maior do que o especificado através dos argumentos de entrada do programa.

2.4. Modularização

Para garantir uma melhor qualidade de código e facilidade de manutenção, temos o programa como um todo dividido em 9 módulos com obrigações diferentes:

O módulo **lexicographic-analyser** (é a classe responsável por realizar as regras de negócio para gerar os resultados da análise lexicográfica através dos dados de entrada).

O módulo **lexicographic-util** (é o contexto responsável por fornecer métodos utilitários para ajudar no processamento das regras de negócio do **lexicographic-analyser**, como é o caso do método de comparação lexicográfica entre duas palavaras).

O módulo **shared-util** (é o contexto responsável por fornecer métodos utilitários compartilhados entre todos os módulos existentes).

O módulo **arrangement-list** (é a estrutura de dados de lista utilizada pelo **lexicographic-analyser** para gerenciar todos os modelos durante a execução do programa).

O módulo **arrangement-list-sorting** (é a classe responsável por realizar todo o processo de ordenação utilizado na estrutura de dados **arrangement-list**).

O módulo **memlog** (realiza a gravação de acessos de memória durante a execução do programa)

O módulo **app** (funciona como um orquestrador do programa, fazendo a leitura dos dados de entrada e utilizando o **poker-face** para processar esses dados e gerar um arquivo de saída).

O módulo **app-validation** (é o contexto responsável por fornecer métodos que validem os dados que são utilizados durante a execução do **app**).

O módulo **app-util** (é o contexto responsável por fornecer os métodos utilitários que são utilizados durante a execução do **app**).

3. Análise de Complexidade

A análise de complexidade do programa foi realizada utilizando como base o módulo **app**, que é o ponto de entrada de execução de todas as regras de negócio.

3.1. Tempo

Dentro do módulo do **app**, ocorre a execução de um loop de leitura e dos seguintes métodos (que podemos considerar como os mais impactantes):

- **LexicographicAnalyser.readOrdering;**
- **LexicographicAnalyser.readWord;**
- **LexicographicAnalyser.getResult**.

Ou seja, vamos iniciar analisando a complexidade de cada um desses métodos para depois analisar a complexidade do módulo **app** como um todo.

No método **LexicographicAnalyser.readOrdering**, todo o conjunto de operações executados nos dão uma complexidade $O(1)$. Ou seja, nesse método temos a seguinte complexidade:

Melhor caso = Pior Caso = $O(1)$

No método **LexicographicAnalyser.readWord**, as ações executadas de maior custo são: formatamos a palavra $O(n)$, verificamos se essa palavra já foi computada anteriormente $O(n)$, atualizamos ou criamos uma nova informação sobre a ocorrência da palavra $O(1)$ / $O(n)$.

Ou seja, teremos a seguinte complexidade para este método:

Melhor caso = Pior caso = $O(n)$

No método **LexicographicAnalyser.getResult**, as ações executadas de maior custo são: copiar as palavras lidas para uma nova lista $O(n)$, criar o resultado em uma lista com a ocorrência de palavras seguindo a nova ordem lexicográfica $O(n)$. Por último, uma das nossas operações mais custosas nesse método é a de ordenação. Portanto, vamos tratar ela de forma mais específica logo abaixo.

Essa operação de ordenação faz uso do algoritmo QuickSort, porém dependendo do tamanho de partição máximo fornecido pelos argumentos de inicialização do programa ele pode usar o algoritmo SelectionSort nessa partição.

Normalmente esses algoritmos possuem uma ordem de complexidade já conhecida, entretanto, fazemos uma operação de custo fora do padrão dentro deles que é a comparação utilizando a ordem lexicográfica $O(1)$ / $O(n)$.

Dessa forma, quando utilizamos o QuickSort na ordenação, temos uma complexidade $O(log\ n)$ / $O(n^3)$. Por fim, caso utilizarmos o SelectionSort podemos ter uma complexidade $O(n^2)$ / $O(n^3)$

Ou seja, podemos concluir que nesse método temos a seguinte complexidade:

Melhor caso = $O(log\ n)$

Pior caso = $O(n^3)$

Agora que analisamos cada um dos métodos, podemos voltar para a análise do módulo **app**.

Temos um loop inicial de n-repetições com as seguintes ações: LexicographicAnalyser.readOrdering $O(1)$ e LexicographicAnalyser.readWord $O(n)$.

Ou seja, nessa primeira parte do módulo **app** temos a seguinte complexidade:

Melhor caso = Pior caso = $O(n)$

Fora do loop, nós executamos as seguintes ações: LexicographicAnalyser.getResult $O(log\ n)$ / $O(n^3)$, salvamos os resultados de cada rodada $O(n)$.

Ou seja, na segunda parte do módulo **app** temos a seguinte complexidade:

Melhor caso = $O(n)$

Pior caso = $O(n^3)$

Ou seja, podemos concluir que a nossa complexidade temporal para o módulo do **app** como um todo pode ser dada por:

Melhor caso = $O(n)$

Pior caso = $O(n^3)$

3.2. Espaço

Como a única parte recursiva do nosso programa é quando usamos o algoritmo de ordenação QuickSort, ele tem um grande impacto na complexidade de espaço.

Ou seja, podemos concluir que a complexidade espacial do nosso programa como um todo vem do uso do QuickSort, que é dada por:

$$Complexidade\ Espacial\ =\ O(n)$$

4. Estratégia de Robustez

Quando estamos lidando com textos, não é novidade que o formato das palavras podem vir nos formatos mais inesperados e diversos possível, o que pode gerar alguns problemas durante a execução do programa.

Nesse sentido, durante a leitura das palavras do texto, temos o cuidado de salvar seu valor original e também de utilizar um método para formatar a palavra no formato que ela será mostrada ao final do programa (em caixa baixa e sem caracteres especiais).

Por fim, foram realizadas validações durante a execução principal do programa para garantir que o arquivo de entrada existe e também que o arquivo de saída foi gerado com sucesso.

5. Testes

Foi realizada uma série de testes para garantir que todas as operações estão sendo executadas corretamente. Logo abaixo é possível ver uma entrada que foi utilizada em um dos testes:

- Entrada:
#ORDEM
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
#TEXTO
Era uma vez UMA gata xadrez

Com a entrada acima, fomos capazes de obter a saída esperada.

- Saída
xadrez 1
vez 1
uma 2
gata 1
era 1
#FIM

6. Análise Experimental

Agora, podemos efetuar uma análise de desempenho computacional e eficiência de acesso à memória.

6.1. Desempenho Computacional

Para esse experimento, foram utilizados textos de 1000, 2000, 3000, 4000 e 5000 palavras (não necessariamente diferentes), com os seguintes argumentos de configuração do QuickSort:

- Mediana = 1
- Tamanho Máximo da Partição = Quantidade de Palavras - 50

Vamos usar a métrica de tempo de execução de cada operação como métrica relevante para a avaliação do nosso programa, visto que é uma métrica que nos possibilita entender o custo de cada execução na prática e fazer uma comparação com a análise de complexidade que fizemos anteriormente. Sendo assim, o tempo de execução (em segundos) para cada quantidade de rodadas pode ser visto na tabela abaixo:

Quantidade de Palavras	**Tempo de execução** *(segundos)*
1000	0.162840853
2000	0.336137565
3000	0.494526265
4000	0.655980390
5000	0.684655469

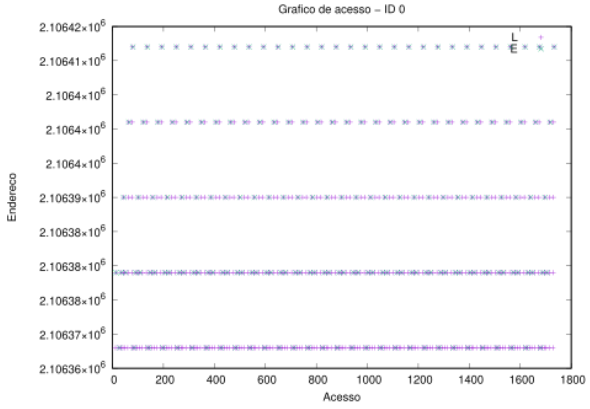
Conforme vimos na nossa análise de complexidade, o melhor caso do nosso programa possui complexidade linear. Sendo assim, ao analisar os dados adquiridos acima, é possível entender que os arquivos de entrada utilizados conseguiram fazer uso do melhor caso do algoritmo, visto que houve um crescimento linear no tempo de execução.

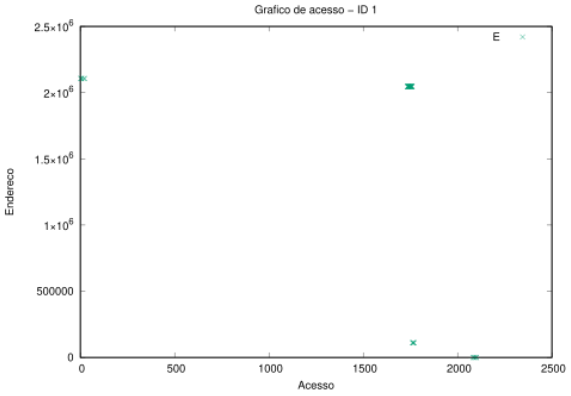
Entretanto, isso não é um comportamento garantido para todos os tipos de entrada, visto que em alguns momentos podemos ter nosso pior caso - que é não linear.

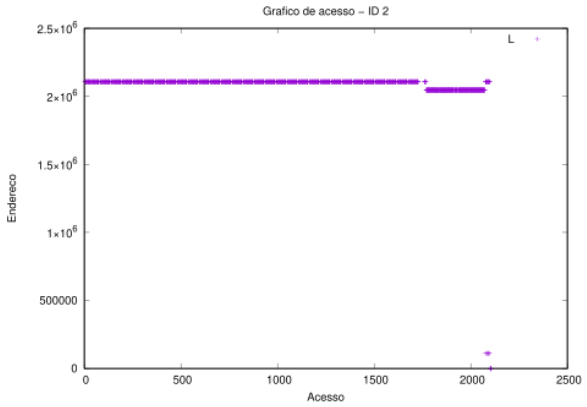
Por fim, ao utilizar o **gprof** para analisar cada operação efetuada em um texto de 5000 palavras, é possível entender que o maior tempo de execução vem do método **LexicographicAnalyser.getResult**, mesmo que ele seja executado apenas uma vez. Isso era esperado pois, conforme vimos na análise de complexidade do programa, esse método é o que possui a maior ordem de complexidade.

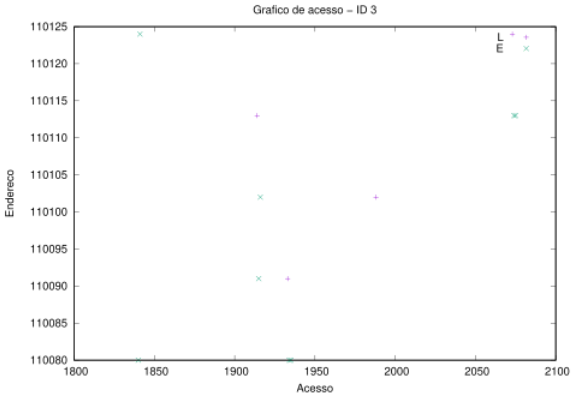
Nesse sentido, sabemos que esse custo ocorre por conta da comparação lexicográfica, ou seja, podemos fazer uma otimização nesse método de comparação para melhorar sua ordem de complexidade e, dessa forma, melhorar também o desempenho do programa como um todo.

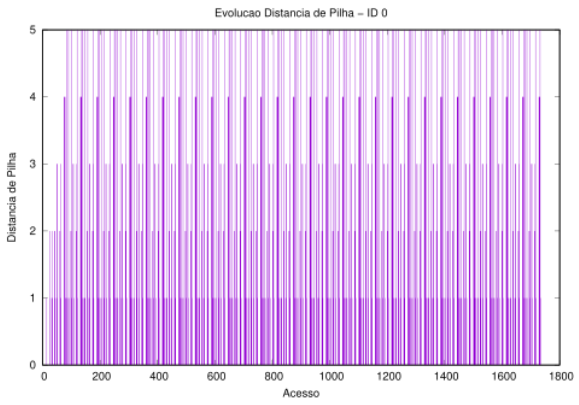
6.2. Análise de padrão de Acesso à Memória e Localidade de Referência

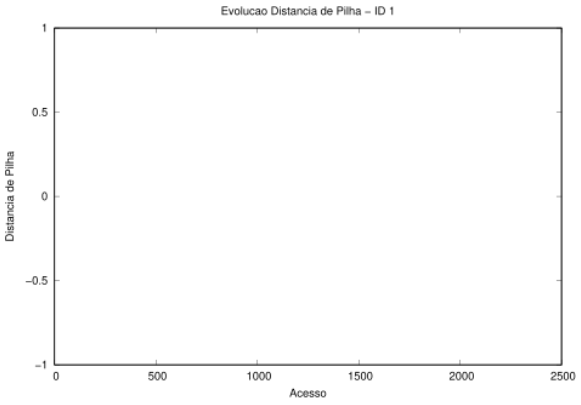


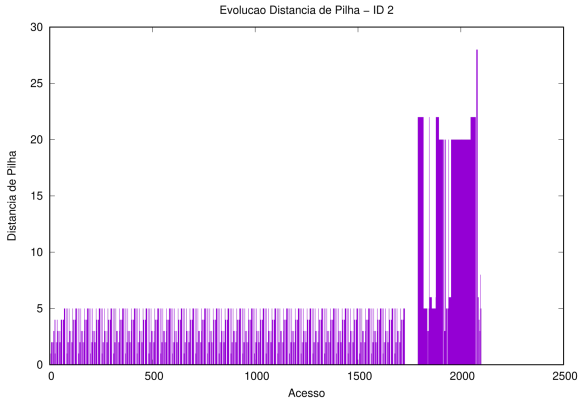


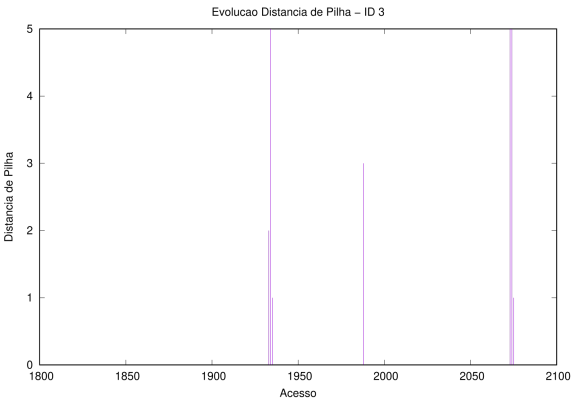


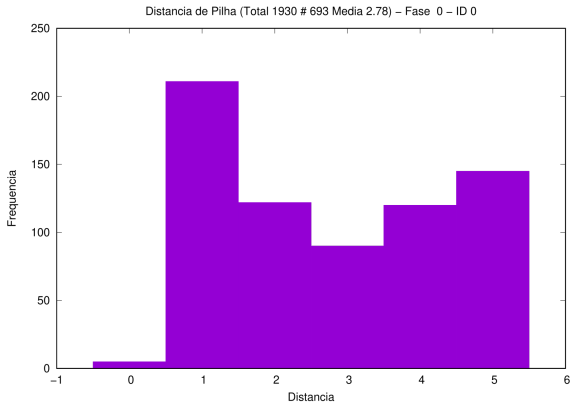


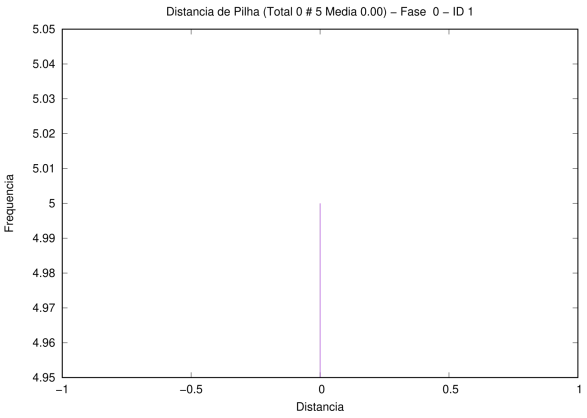


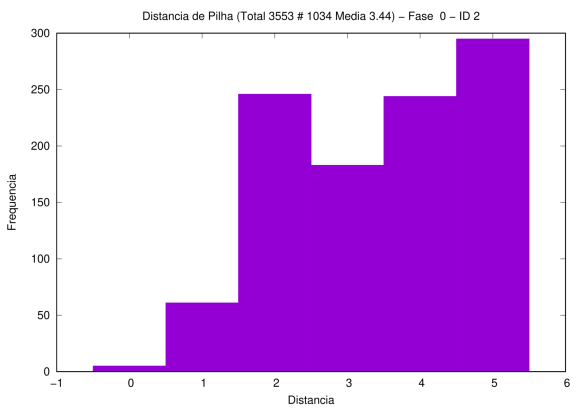


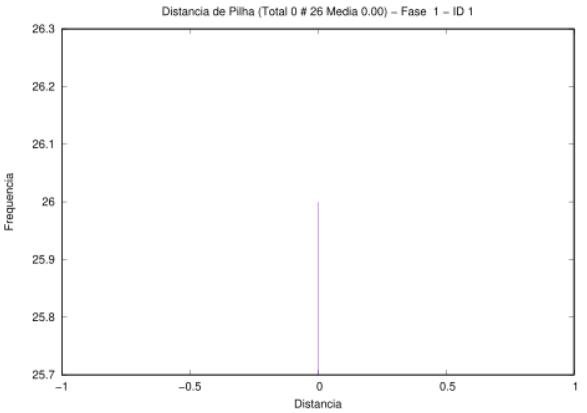


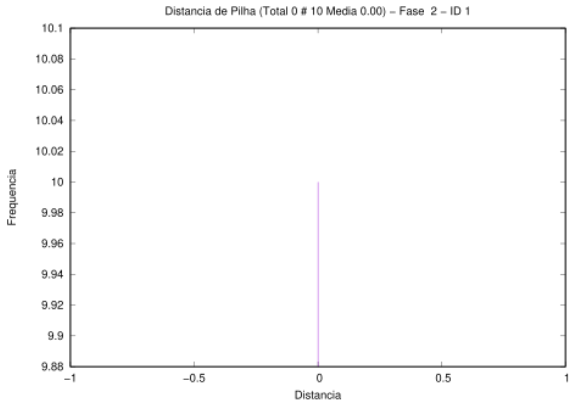


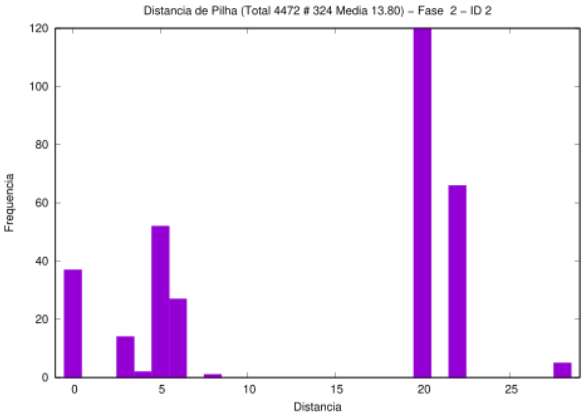

Distancia de Pilha (Total 1930 # 693 Media 2.78) – Fase 0 – ID 0
Frequencia
Distancia

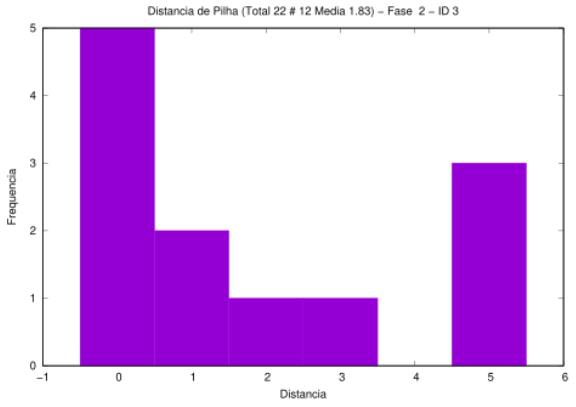

Distancia de Pilha (Total 0 # 5 Media 0.00) – Fase 0 – ID 1
Frequencia
Distancia


Distancia de Pilha (Total 3553 # 1034 Media 3.44) – Fase 0 – ID 2
Frequencia
Distancia


Distancia de Pilha (Total 0 # 26 Media 0.00) – Fase 1 – ID 1
Frequencia
Distancia


Distancia de Pilha (Total 0 # 10 Media 0.00) – Fase 2 – ID 1
Frequencia
Distancia


Distancia de Pilha (Total 4472 # 324 Media 13.80) – Fase 2 – ID 2
Frequencia
Distancia


Distancia de Pilha (Total 22 # 12 Media 1.83) – Fase 2 – ID 3
Frequencia
Distancia

De início, segue abaixo o que cada ID e Fase significam nos gráficos mostrados acima:

Fases

- Fase 0: Leitura de palavra do texto
- Fase 1: Leitura de caractere da nova ordem lexicográfica
- Fase 2: Cálculo do resultado final

ID

- ID 0: Atualização de informação da lista
- ID 1: Criação de informação na lista
- ID 2: Pesquisa de informação na lista
- ID 3: Ordenação utilizando QuickSort
- ID 4: Ordenação utilizando SelectionSort

Nos gráficos de acesso durante operações de atualização e pesquisa de informações da lista, é possível perceber uma padronização de acesso. Isso ocorre visto que durante essas operações, nós buscamos um item na lista, dessa forma, como a nossa estrutura de dados é um vetor estático, as posições de memória se mantêm próximas. Durante as operações de criação só temos apenas informações de escrita visto que nenhum acesso precisa ser efetuado durante este momento.

Além disso, é possível notar que durante a operação de atualização de informação da lista, nossa distância de pilha mantêm um valor baixo, visto que quando estamos lendo uma palavra, as chances dela já ter sido computada é baixa (pois um texto geralmente possui várias palavras diferentes), ou seja, é mais comum da palavra ser criada na nossa estrutura de dados ao invés de ser atualizada. Dessa forma, como na operação de criação ocorre apenas uma busca sequencial e padronizada de todos os elementos, mantemos uma distância de pilha mais baixa.

No entanto, é possível notar que há um aumento relevante da distância de pilha quando estamos efetuando as últimas pesquisas de palavras nas nossas estruturas de dados. Isso é um comportamento que começa durante nosso processo de ordenação, visto que durante esse processo, é necessário fazer uma comparação lexicográfica de duas palavras utilizando uma estrutura de dados que guarda os caracteres da nossa nova ordem lexicográfica. Ou seja, já que para cada conjunto de palavras comparadas, é necessário comparar cada letra de cada uma das palavras, nós temos praticamente um comportamento aleatório para encontrar o valor de ordem lexicográfica de cada letra, gerando então uma distância de pilha mais elevada.

Podemos ver que, no geral, é possível manter uma distância de pilha bem pequena durante a parte inicial do programa (leitura de ordem lexicográfica e de palavras do texto), porém, a distância de pilha se torna mais alta durante o momento de ordenação, visto que fazemos diversas leituras quase que “aleatórias”.

7. Conclusões

Nesse trabalho prático, fizemos um programa que recebe um texto no qual realizamos uma contagem de ocorrências de cada palavra, além de uma ordem lexicográfica que é usada para ordenar o resultado que é mostrado ao final da execução. Como estamos lidando com texto e geralmente o conteúdo dele pode estar formatado da maneira mais inesperada possível, foi necessário fazer uma formatação correta de cada palavra encontrada. Em seguida, através da execução desse programa, conseguimos gerar dados de desempenho computacional Por fim, fomos capazes de tratar os dados gerados e analisá-los para entender o comportamento de forma mais aprofundada.

Diante disso, podemos compreender que apesar de podermos escolher uma classe de algoritmos de ordenação excelente para resolver um problema, nós somos capazes de mudar o desempenho desse algoritmo (seja para melhor ou pior) dependendo de como as comparações das informações são feitas durante a ordenação (como por exemplo a comparação de ordem lexicográfica).

Por fim, é importante citar que uma das maiores dificuldades nesse trabalho surgiu no sentido de conseguir realizar a comparação lexicográfica da maneira correta durante a ordenação das palavras.

Apêndice

Instruções para compilação e execução

Antes de efetuar a compilação do programa principal, é necessário abrir a pasta **analisamem** e rodar o comando **make**.

Agora, para efetuar a compilação do programa, basta entrar na pasta **TP** e rodar o comando **make** em seu terminal.

Esse comando irá executar as tarefas presentes no arquivo **Makefile** e efetuar tanto a compilação quanto a execução do programa.