Trabalho Prático **2**

Documentação

Estrutura de Dados 2022/01

Bernardo Reis de Almeida (2021032234)

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais
Belo Horizonte - MG - Brasil
bera01@ufmg.br

1. Introdução

O **tratamento de textos** é um dos problemas mais relevantes na área da computação. Desde tradutores e mecanismos de busca, a processamento de dados e informações ou aprendizagem de máquina, são diversas as suas aplicações e inquestionáveis sua relevância e ubiquidade. Frente a isso, o presente trabalho prático tem por objetivo a aplicação de conceitos relacionados a **estruturas de dados** e a **algoritmos de ordenação** sob pretexto de um contato com esse campo de estudos e com suas possibilidades.

Em especial, foi exigida a elaboração de um programa responsável por identificar a ocorrência de palavras em um texto, tratá-las com base em critérios de formatação pré-estabelecidos e as ordenar conforme um parâmetro arbitrário de precedência. Um arquivo de entrada é fornecido contendo uma ordem lexicográfica e um conjunto de palavras. O programa deve, a partir daí, tratar adequadamente esse conjunto, contabilizar a quantidade de aparições de cada palavra distinta e retornar de maneira ordenada essas informações por meio de um arquivo de saída.

A abordagem adotada para a resolução desse problema pode ser separada em duas principais etapas. Inicialmente, os dados fornecidos são armazenados em memória e formatados conforme os critérios definidos. Em seguida, uma **estrutura de dados** - nomeadamente uma **lista** - é utilizada para a **organização**, para a **contagem** e para a **ordenação** do conjunto formatado de palavras. Esta é, em uma terceira etapa, percorrida sequencialmente e suas informações, exportadas à saída do programa.

Inicialmente, os detalhes acerca de decisões de projeto e de implementação serão discutidos na seção **[Método]**. Em seguida, a complexidade computacional em termos de tempo e de espaço dos algoritmos elaborados, formalizada por uma notação matemática assintótica, será tratada na seção **[Análise de Complexidade]**. As estratégias de manutenção da integridade e da robustez da aplicação, por sua vez, serão apresentadas e justificadas na seção **[Estratégias de Robustez]**. A seção **[Análise Experimental]** é dedicada aos resultados de testes realizados acerca do desempenho computacional e da localidade de referência, assim como a uma breve discussão sobre as informações coletadas. Por fim, um sumário sobre o processo de desenvolvimento como um todo, as fontes bibliográficas utilizadas e as instruções para a geração e para a execução do programa podem ser encontrados, respectivamente, nas seções **[Conclusões]**, **[Bibliografia]** e **[Instruções para Compilação e Execução]**.

2. Método

a. Ambiente Computacional

O programa foi desenvolvido em **linguagem C++** (versão 11), em um ambiente computacional **Linux** virtualmente simulado no sistema operacional Windows (Debian GNU/Linux 11 (bullseye) on Windows 10 x86_64) (5.10.16.3-microsoft-standard-WSL2). O compilador utilizado foi o **G++** (versão (Debian 10.2.1-6) 10.2.1

20210110). Os testes foram executados em uma máquina com um processador de 3.60 GHz (6 núcleos, 12 threads) e com 16 gigabytes de memória RAM.

b. Implementação

A abordagem adotada - em termos de implementação - segregou o problema enunciado em duas principais subetapas: a **extração** e o **tratamento** da entrada de dados conforme regras de formatação pré-estabelecidas e sua **contabilização** e **ordenação**, novamente, sob um critério de precedência arbitrário.

A primeira etapa foi desenvolvida por meio de procedimentos auxiliares que materializam as operações de extração e de formatação. As declarações desses procedimentos estão contidas em um arquivo de cabeçalho dedicado, vulgo **stringOperations.hpp**. Em particular, arquivos de entrada e de saída de dados são fornecidos via linha de comando. Dentro daquele, estão contidos o texto a ser tratado e o critério de precedência a ser utilizado, ambas informações que devem ser devidamente formatadas e armazenadas para uso pelo programa. A esse pretexto, houve a definição de três procedimentos:

- **gatherArguments():** Recebe como parâmetros os argumentos da linha de comando - contidos no vetor argv, definido pela linguagem C++ -, sua quantidade - variável inteira - e referências para as variáveis que os armazenarão em memória. Em especial, o programa reconhece quatro argumentos como válidos (detalhes na seção **[Instruções para Compilação e Execução]**) , sendo estes os nomes dos arquivos de entrada e de saída - variáveis do tipo string -, e dois parâmetros para a função de ordenação da estrutura de dados (detalhes adiante) - variáveis inteiras. O procedimento itera sobre todos os argumentos, verificando a presença dos identificadores válidos. Na ocasião da existência de algum, seu valor é devidamente atribuído à variável correspondente.

- **gatherTextAndOrder():** Recebe como parâmetros o arquivo de entrada e referências para variáveis que armazenarão os dados relativos à precedência, ao texto - variáveis do tipo stream - e à quantidade de palavras nele contidas - variável inteira. O arquivo de entrada é percorrido e suas informações são apropriadamente direcionadas à variável correspondente - texto ou ordem. A variável relativa à quantidade de palavras é progressivamente incrementada a cada leitura da entrada de dados direcionada ao texto, sendo o procedimento realizado palavra por palavra.

- **formatText():** Recebe como parâmetro uma referência para a variável - do tipo stream - que contém o texto, o qual é progressivamente corrigido conforme critérios de formatação pré-estabelecidos. Em particular, essa correção é feita palavra por palavra, caractere a caractere, cada qual sendo analisado quanto à conformidade às regras definidas e devidamente corrigido, caso aplicável.

A segunda etapa, por sua vez, é baseada em uma estrutura de dados - uma **lista sequencial** - responsável por armazenar as palavras contidas no texto em conformidade com as necessidades de contagem e de ordenação. Sob essas condições, ela opera armazenando cada palavra única e sua respectiva quantidade de aparições, além de que se pauta na premissa de que todos os dados fornecidos estarão formatados conforme critérios pré-estabelecidos. Novamente, essa estrutura foi definida em um arquivo de cabeçalho dedicado, seja ele **stringList.hpp**. A escolha de uma lista sequencial se deve principalmente ao fato de o tratamento previamente realizado sobre o texto permitir se ter conhecimento do número máximo de palavras nele contidas, o que torna viável uma alocação estática da estrutura mencionada. Além disso, esse modelo de implementação não envolve apontadores explícitos e apresenta um acesso direto via índice, o que,

além de ser econômico em termos de memória, facilita a leitura e a troca de elementos. Tendo em vista que o programa irá executar principalmente operações de acesso, de inserção e de ordenação, todas essas características o favorecem positivamente. A lista foi modelada por meio de duas classes:

- **StringNode:** Representa uma célula da lista. Contém dois atributos, um referente à palavra em si - variável do tipo string - e outro referente a sua quantidade de aparições - variável inteira. Um construtor sobrecarregado inicializa um objeto de duas maneiras diferentes: caso uma palavra seja passada como parâmetro, esta é armazenada no atributo correspondente e a quantidade é iniciada com o valor 1; caso contrário, um objeto "inválido" é instanciado, contendo uma string vazia e uma quantidade nula. Este último caso é estritamente utilizado para a alocação inicial da lista, sendo cada célula efetivamente em uso inicializada apropriadamente. Além disso, dois métodos são definidos:

    - **biggerThan() e smallerThan():** Responsáveis por comparar duas células entre si. Ambos recebem uma referência a um objeto da classe e a um vetor de ordenação (detalhado adiante) e retornam um valor booleano relativo à condição de precedência - maior que e menor que, respectivamente - do objeto sobre o qual são invocados com relação àquele passado como parâmetro.

  Todos os membros da classe são privados e sua operação é restrita à classe StringList, tendo em vista a abstração desta como um conjunto de palavras e aqueles serem apenas mecanismos internos.

- **StringList:** Representa a lista em si. Contém atributos privados referentes ao tamanho atual da lista, ao seu tamanho máximo, ao índice do primeiro elemento, ao índice do último elemento - variáveis inteiras -, à lista em si - ponteiro do tipo StringNode - e ao vetor de ordenação - ponteiro do tipo inteiro. O vetor de ordenação consiste em uma estrutura com 26 entradas, cada qual relativa a uma letra do alfabeto da língua portuguesa e cujo valor diz respeito ao peso da referida letra sob o critério de precedência estabelecido. Um construtor recebe o tamanho máximo da lista e instancia um objeto da classe, atribuindo tal parâmetro ao atributo correspondente, definindo o tamanho atual como 0 e estabelecendo os índices do primeiro e do último elementos como valores nulos (já que a lista está vazia). A lista em si, modelada por um vetor de células, é alocada com o tamanho máximo. O vetor de ordenação também é alocado e inicialmente assume pesos correspondentes à ordem lexicográfica padrão. Um destrutor contém comandos para a desalocação da memória alocada ao término da vida do objeto. Ainda, quatro métodos públicos são definidos:

    - **insert():** Recebe uma palavra como parâmetro. O método percorre a lista para verificar se a palavra já se encontra nela, incrementando sua quantidade de aparições se for o caso. Ao contrário, insere a palavra ao final e atualiza o índice do último elemento.

    - **setOrder():** Recebe uma entrada de dados - sob a forma de uma variável do tipo stream - com informações relativas a um critério de precedência. A entrada é lida e os pesos do vetor de ordenação do objeto são atualizados de maneira conforme.

    - **order():** Recebe dois parâmetros de operação - variáveis inteiras - e ordena a lista de acordo com o critério de precedência estabelecido. A ordenação é feita com base nos algoritmos

Quick Sort e Insertion Sort e envolve quatro métodos auxiliares - **quickSort()** e **partition()** (implementam o Quick Sort), **insertionSort()** (implementa o Insertion Sort) e **median()** - todos privados. Um dos parâmetros diz respeito ao número de elementos a serem considerados no cálculo da mediana para a determinação do pivô - método partition() chama o método median() para a determinação da mediana dos 'x' primeiros elementos, sendo 'x' o valor passado como parâmetro. O outro se refere à quantidade de elementos em uma partição a partir da qual o algoritmo Insertion Sort deve ser utilizado ao invés do Quick Sort - o método quickSort() passa a chamar o método insertionSort() ao invés do partition() para a ordenação das partições a partir de determinado tamanho 'y', sendo 'y' o parâmetro passado. Por padrão - caso os parâmetros não sejam fornecidos via linha de comando ou não sejam válidos -, a mediana retorna o primeiro elemento da partição e apenas o algoritmo Quick Sort é inteiramente utilizado para a ordenação, comportamento este que pode ser alterado, como já mencionado. Um último detalhe é que a função median() **não ordena a estrutura de dados**, logo, não há conflitos entre ambos os parâmetros. Caso a partição tenha menos elementos do que o determinado para o cálculo da mediana, o primeiro será retornado. Caso a lista tenha menos elementos do que o determinado para o uso do Insertion Sort, este algoritmo será inteiramente utilizado para sua ordenação.

- **print():** Recebe um arquivo de saída como parâmetro. O método percorre todas as células válidas da lista e exporta a esse arquivo as informações de cada uma, a saber, a palavra e sua quantidade de aparições no texto.

Finalmente, a função **main()**, contida no arquivo **main.cpp**, é responsável por proceder com ambas as etapas e garantir uma correta transição entre elas. Inicialmente, as variáveis de controle que irão receber os parâmetros passados pela linha de comando são declaradas e inicializadas por meio do procedimento gatherArguments(). Em seguida, os arquivos de entrada e de saída são devidamente abertos e as variáveis que armazenarão as informações relativas ao texto e ao critério de precedência são definidas e inicializadas mediante o procedimento gatherTextAndOrder(). O texto é tratado a partir de uma chamada ao procedimento formatText() e, após isso, um objeto da classe StringList é criado e populado com o conteúdo nele contido, isto é, para cada palavra, o método insert() é invocado para sua contabilização. O atributo de ordenação do objeto também é adequadamente definido por uma chamada ao método setOrder(). A lista é, então, ordenada pelo método order() e exportada ao arquivo de saída pelo método print().

3. Análise de Complexidade

O programa apresenta três peculiaridades com relação a sua complexidade computacional. Em primeiro lugar, alguns parâmetros podem ser modificados de modo a alterar o comportamento de sua função de ordenação, sendo esta uma das mais relevantes em termos de custo de execução. Esta peculiaridade será abordada em detalhes adiante, na análise dos métodos correspondentes. Em segundo lugar, diferentes algoritmos apresentam custos escaláveis com diferentes elementos da entrada de dados. Em especial, por se tratar de um texto, há operações que atuam iterativamente sobre palavras e, portanto, dependem de um valor, mas há também operações que atuam iterativamente sobre caracteres, dependendo, portanto, de outro valor. Para efeito de simplificação, será definido que a quantidade de palavras no texto será dada pela variável 'n' e que a quantidade média de caracteres em cada palavra será um valor constante. Entretanto, reitera-se que, na prática, a quantidade de caracteres por palavra influencia a complexidade do programa, tanto em termos de

tempo, quanto de espaço. Em terceiro lugar, a quantidade de palavras efetivamente armazenadas na lista pode ser menor do que a quantidade de palavras extraídas do arquivo de entrada, já que palavras iguais ocuparão uma mesma célula. Assim, os algoritmos que atuam sobre a estrutura de dados serão escalados por um valor variável 'm', sendo este o tamanho da lista, o qual pode ser menor ou igual a 'n'.

- **gatherArguments():** O procedimento itera sobre todos os argumentos passados pela linha de comando e realiza uma série de operações constantes. Tendo em vista que apenas uma quantidade finita é válida, o número máximo de argumentos que podem ser passados foi limitado, de modo que assuma um caráter aproximadamente constante. Assim, a ordem de complexidade assintótica em termos de **tempo** é **O(1)**. Com relação ao **espaço**, todos os parâmetros são passados por referência e, internamente, apenas variáveis constantes e um vetor de tamanho finito e não escalável são criados, de modo que a ordem de complexidade também seja **O(1)**.

- **gatherTextAndOrder():** O procedimento itera sobre todas as palavras contidas no arquivo de entrada e as armazena apropriadamente nas variáveis referentes ao texto e ao critério de precedência. Como o número de palavras neste último é constante, o método depende apenas da quantidade 'n' de palavras no texto. Assim, sua ordem de complexidade assintótica em termos de **tempo** é **O(n)**. Em termos de **espaço**, os parâmetros são recebidos por referência e são criadas apenas algumas variáveis constantes, de modo que sua ordem de complexidade seja **O(1)**.

- **formatText():** O procedimento itera sobre todas as palavras contidas no texto a fim de se aplicar correções conforme os critérios de formatação pré-estabelecidos. Assim, no total, são realizadas 'n' operações constantes, de modo que a ordem de complexidade assintótica em termos de **tempo** seja **O(n)**. Em termos de **espaço**, o procedimento cria uma variável local de tamanho proporcional à quantidade de palavras no texto, de modo que sua ordem de complexidade também seja **O(n)**.

- **biggerThan() e smallerThan():** Ambos os métodos comparam duas células da lista por meio de operações de custo constante. Dessa forma, sua ordem de complexidade assintótica em termos de **tempo** é **O(1)**. Em termos de espaço, os métodos envolvem apenas variáveis constantes e parâmetros recebidos por referência, de modo que sua ordem de complexidade também seja **O(1)**.

- **insert():** O método percorre a lista de palavras para verificar a condição de existência daquela a ser inserida, executando operações constantes no caso dela ser ou não encontrada. Dessa forma, sua complexidade pode ser enquadrada em diferentes casos, a depender da organização dos dados. Em especial, sua ordem de complexidade assintótica em termos de **tempo** irá apresentar um **melhor caso O(1)** - dado pela palavra estar contida na primeira posição da lista - e um **pior caso O(m)** - dado pela palavra estar na última posição da lista ou não estar nela. Em termos de **espaço**, o método envolve algumas variáveis constantes e a própria estrutura de dados em si, de modo que sua ordem de complexidade seja **O(m)**.

- **setOrder():** O método executa um conjunto de operações sobre a variável que armazena o critério de precedência. Como o tamanho desta é sempre constante e não escalável, a ordem de complexidade assintótica geral em termos de **tempo** é **O(1)**. Em termos de **espaço**, apenas algumas variáveis auxiliares constantes são criadas, de modo que sua ordem de complexidade também seja **O(1)**.

- **order():** O método realiza a ordenação da lista com base em um balanço entre a execução dos métodos quickSort() e insertionSort(). Considerando uma situação genérica de uso e uma estrutura de dados de tamanho 'x', estes últimos apresentam os seguintes comportamentos assintóticos:
    - **quickSort():** O funcionamento deste método é baseado na divisão do problema de se ordenar a lista em subproblemas, os quais são então parcialmente resolvidos e novamente subdivididos progressivamente até o ordenamento total da estrutura de dados. Em particular, a lista será inicialmente processada pelo algoritmo, o qual percorrerá seus 'x' elementos e executará uma forma de ordenação parcial baseada em um deles, nomeadamente o pivô. A partir daí, ela será dividida em duas partições de tamanho em média similar, sobre as quais o mesmo procedimento será realizado, de modo que todos os 'x' elementos sejam novamente percorridos, mas agora cada qual em sua partição. Esse processo é repetido até que não haja mais a possibilidade de se subdividir a lista, o que ocorre, em média, após log(x) repetições. Dessa forma, a ordenação total levará log(x) etapas, dentro de cada qual são realizadas 'x' operações constantes. Assim, a ordem de complexidade assintótica do algoritmo em termos de **tempo** é **O(x*log(x))**. Esse resultado, novamente, é uma média, dado que o algoritmo como um todo apresenta uma série de diferentes casos a depender principalmente da escolha do pivô. Em particular, caso o pivô seja o menor ou o maior elemento em todas as partições, o algoritmo assume um comportamento quadrático. Para minimizar as chances de ocorrência dessa situação, o pivô é escolhido como a mediana dos 'y' primeiros elementos da lista, sendo 'y' um argumento passado ao programa. Esse cálculo é realizado por meio do método **median()**, o qual compara esses elementos entre si a fim de se determinar o mediano. Em termos de complexidade **temporal**, caso a mediana seja o primeiro elemento - **melhor caso** -, a ordem será **O(y)**, enquanto que, caso seja o último - **pior caso** -, a ordem será **O(y^2)**. Em termos de **espaço**, tanto a função quickSort(), quanto a função median(), envolvem algumas variáveis constantes e a própria estrutura de dados em si, de modo que sua ordem de complexidade seja **O(x)**.

    - **insertionSort():** O método percorre cada elemento da estrutura de dados e o compara com os anteriores, movimentando-o até que assuma uma posição correta com relação a essa partição da lista, de modo que, ao final, todos estejam ordenados. Dessa forma, há dois casos de execução, a depender da organização da entrada de dados. Em um melhor caso, dado por uma lista ordenada, cada elemento já estará na posição relativa correta, o que pode ser verificado com apenas operações constantes para cada. Em um pior caso, dado por uma lista inversamente ordenada, cada elemento deverá ser operacionalizado uma quantidade de vezes proporcional à sua posição na lista. Assim, a ordem de complexidade assintótica do método em termos de **tempo** é dada por, em um **melhor caso**, **O(x)** e, em um **pior caso**, **O(x^2)**. Em termos de **espaço**, o algoritmo envolve algumas variáveis constantes e a própria estrutura de dados em si, de modo que sua ordem de complexidade seja **O(x)**.

No caso, a complexidade computacional do método order() irá depender dos parâmetros de partição e de mediana, assim como da organização dos dados e da operação dos métodos auxiliares de ordenação. Entretanto, de maneira geral, em termos de **tempo**, há um **pior caso** de execução - o qual pode ocorrer em qualquer um dos piores casos de cada um dos métodos auxiliares -, dado pela ordem de complexidade assintótica **O(m^2)**, e um **melhor caso** de execução - Insertion Sort e

mediana são utilizados de maneira constante (não escalável com a entrada de dados) e o Quick Sort não recai sobre o pior caso -, dado por **O(m*log(m))**. Em termos de **espaço**, assim como seus métodos auxiliares, o método order() trabalha apenas com algumas variáveis constantes e com a estrutura de dados em si, de modo que sua ordem de complexidade seja **O(m)**.

- **print():** Este método itera sobre todos os elementos contidos na lista, exportando suas respectivas informações a um arquivo de saída. Assim, sua ordem de complexidade assintótica em termos de **tempo** é dada por **O(m)**. Em termos de espaço, o método envolve apenas algumas variáveis constantes, além da estrutura de dados em si, assumindo uma ordem de complexidade **O(m)**.

O programa, em sua totalidade, realiza chamadas, direta e indiretamente, a todos os métodos, funções e procedimentos enumerados, de modo que sua complexidade seja conforme a eles. Retomando o funcionamento da função main(), a ordem de complexidade geral pode ser subdivida como um resultado de três sub etapas: a **extração das informações da entrada** - a qual escala de maneira linear com a quantidade 'n' de palavras no texto -, a **população da lista** - para cada palavra, a lista é percorrida, de modo que haja margem para um comportamento quadrático - e a sua **operacionalização** - regida principalmente pelo método de ordenação. De uma maneira generalizada, relevando todas as possíveis excepcionalidades, é possível definir que a ordem de complexidade **temporal** do programa apresenta os seguintes casos:

- **Pior caso O(n^2):** Caracterizado por todas as palavras contidas no arquivo de entrada serem distintas entre si. Nesse caso, a lista irá possuir um tamanho equivalente à quantidade de palavras no texto de entrada ('m' = 'n'). A etapa de extração de informações irá depender da quantidade 'n' de palavras no texto. A população, por outro lado, apresentará complexidade O(n^2). Como cada palavra é distinta entre si, a operação de inserção irá sempre percorrer a lista inteira um quantidade 'n' de vezes (já que uma dada palavra a ser inserida não será igual a nenhuma outra) antes de efetivamente realizar uma inserção, originando um comportamento quadrático. A ordenação da estrutura de dados, por sua vez, apresentará uma complexidade que varia entre O(n) - melhor caso do Insertion Sort - a O(n^2) - pior caso do Quick Sort ou do Insertion Sort -, com um caso médio em O(n*log(n)) - melhor caso do Quick Sort. Independentemente, a ordem de complexidade geral, em função da etapa de inserção, é O(n^2).

- **Melhor caso O(n):** Caracterizado por todas as palavras contidas no arquivo de entrada serem iguais entre si. A etapa de extração, mais uma vez, dependerá da quantidade 'n' de palavras no texto. Nesse caso, entretanto, a lista irá possuir tamanho 1, de modo que todas as operações nela executadas apresentem custo constante, inclusive a ordenação. A inserção de palavras, por sua vez, executará apenas o incremento da quantidade de aparições da palavra única uma quantidade 'n' de vezes, de modo que sua complexidade seja linear. A ordem de complexidade geral, pois, é O(n).

Em termos de **espaço**, o programa como um todo realiza o armazenamento de várias variáveis, das quais as mais relevantes possuem um tamanho que escala com a quantidade de palavras presentes no arquivo de entrada, de modo que sua ordem de complexidade assintótica seja **O(n)**.

4. Estratégias de Robustez

A garantia de robustez e integridade ao funcionamento do programa foi proposta com base no uso da biblioteca **exception**, a qual faz parte da biblioteca padrão da linguagem C++. Em especial, casos de

excepcionalidade em tempo de execução foram abordados mediante o lançamento de exceções, as quais são interpretadas e apropriadamente tratadas pelo programa, a depender da aplicabilidade e do impacto na integralização do algoritmo como um todo. De maneira geral, pode-se subdividir as estratégias adotadas em dois principais ramos: a garantia da **integridade dos dados** e a garantia da **lógica operacional**.

A **integridade dos dados** envolve toda e qualquer manipulação das informações utilizadas pelo programa, desde sua extração de um arquivo de entrada à sua exportação a um arquivo de saída. Em particular, três situações foram relevadas: a passagem de parâmetros por linha de comando, a abertura dos arquivos de entrada e de saída e a leitura/escrita nestes ou em qualquer stream - objeto que foi utilizado como um mediador na comunicação e no tratamento das informações entre as estruturas internas de dados e as fontes externas. Em um primeiro momento, a passagem dos argumentos pela linha de comando é verificada por condicionais que garantem o fornecimento de arquivos de entrada e de saída, assim como de parâmetros válidos, caso estes tenham sido fornecidos. Em seguida, cada ocorrência de uma operação de abertura, leitura ou escrita é avaliada quanto a seu sucesso por uma condicional. Em qualquer um dos casos, a existência de anormalidades implica o lançamento de uma exceção e a interrupção da execução do programa. Esta abordagem é justificada pelo papel fundamental que os dados apresentam no funcionamento do algoritmo como um todo, sendo que qualquer falha nesse quesito o invalidaria ou o instabilizaria profundamente.

A **lógica operacional**, por sua vez, engloba a execução das operações em si, assim como os valores e parâmetros utilizados pelos vários algoritmos pertencentes à parte funcional do programa. Em destaque, duas principais situações foram abordadas: a manipulação da lista sequencial e a passagem de parâmetros para o funcionamento de seus algoritmos de ordenação. Na primeira, qualquer tentativa de manipulação em situações de invalidez - inserção em lista cheia, instanciação de objeto com tamanho máximo negativo/nulo, a não passagem de um vetor de ordenação próprio - emite um aviso e/ou direciona o fluxo de execução para um caso padrão, cada qual quando aplicável. Na segunda, novamente, a passagem de parâmetros inválidos - mediana ou partição entre uma quantidade de elementos maior que o tamanho ou entre uma quantidade negativa/nula - implica o uso de valores padrão. A escolha por essa abordagem se deve ao fato de essas excepcionalidades não necessariamente afetarem a integridade de outras operações que porventura venham a ser executadas em seguida - já que não afetam a integridade interna dos dados -, além de ser possível direcionar sua ocorrência a um fluxo esperado de execução.

5. Análise Experimental

A análise experimental foi realizada por meio da biblioteca **memlog** e da aplicação **analisamem**. Aquela contém diversas funcionalidades para o registro do tempo de execução do programa e de seus acessos à memória, enquanto esta é capaz de plotar gráficos analíticos com base nos resultados da primeira.

a. Desempenho Computacional

A análise de desempenho computacional envolveu a realização de testes que estimularam diferentes casos de execução do programa, originados da variação de seus parâmetros de controle e do tamanho de sua entrada de dados. Como já mencionado, para efeitos de simplificação, o tamanho médio de cada palavra foi tomado como um valor bem definido e constante - no caso, 50 caracteres por palavra. Quatro baterias de experimentos foram realizadas, cada qual avaliando o impacto causado por diferentes alterações nas condições de operacionalização do algoritmo. A seguir, tem-se os resultados obtidos:

Primeiro Teste: Variação na quantidade de palavras, todas iguais entre si. Cada valor utilizado é um múltiplo escalar de um primeiro valor, sendo este o caso base. Os parâmetros de ordenação foram deixados nos valores padrão. Representa a melhor situação de execução do algoritmo, o qual assume uma ordem de complexidade assintótica O(n), sendo 'n' o número de palavras no texto de entrada. Além do resultado prático em si, um resultado teórico é fornecido, sendo este baseado em um cálculo envolvendo a complexidade do algoritmo e o resultado da execução do caso base.

Entrada (n° de palavras)	**Tempo prático**	**Tempo teórico (O(n))**
50000	1.282187996	1.282187996
100000	2.400007628	2.564375992
150000	3.781437837	3.846563988
200000	5.140170873	5.128751984
250000	6.349948115	6.41093998

Segundo Teste: Variação na quantidade de palavras, todas diferentes entre si. Cada valor utilizado é um múltiplo escalar de um primeiro valor, sendo este o caso base. Os parâmetros de ordenação foram deixados nos valores padrão. Representa uma das piores situações de execução do algoritmo (ordenação pelo Quick Sort), em que este assume uma ordem de complexidade assintótica O(n^2), sendo 'n' o número de palavras no texto de entrada. Além do resultado prático em si, um resultado teórico é fornecido, sendo este baseado em um cálculo envolvendo a complexidade do algoritmo e o resultado da execução do caso base.

Entrada (n° de palavras)	**Tempo prático**	**Tempo teórico (O(n^2))**
50000	17.731206303	17.731206303
100000	68.055290472	70.924825212
150000	151.157022669	159.580856727
200000	267.573609975	283.699300848
250000	417.431865706	443.280157575

Terceiro Teste: Variação no parâmetro de mediana, mantendo o parâmetro de partição no caso padrão. Cada valor utilizado é progressivamente incrementado, partindo do valor padrão. A quantidade de palavras é mantida constante - 50000 palavras distintas entre si. Representa o impacto causado na escolha do parâmetro referido na operação do algoritmo como um todo para uma mesma situação de execução.

Parâmetro (n° de elementos na mediana)	**Tempo prático**
1 (padrão)	**17.725342260**
5	17.724240134
50	17.816914304
500	18.277534769
5000	22.171360679
50000	31.419468816

Quarto Teste: Variação no parâmetro de partição, mantendo o parâmetro de mediana no caso padrão. Cada valor utilizado é progressivamente incrementado, partindo do valor padrão. A quantidade de

palavras é mantida constante - 50000 palavras distintas entre si. Representa o impacto causado na escolha do parâmetro referido na operação do algoritmo como um todo para uma mesma situação de execução.

Parâmetro (n° de elementos na partição)	**Tempo prático**
1 (padrão)	**17.746368071**
5	17.671294448
50	17.659846256
500	17.800760510
5000	18.848661834
50000	31.090068677

O primeiro teste almeja verificar o desempenho computacional do programa em seu melhor caso de execução, dado por todas as palavras no texto de entrada serem iguais. Os resultados práticos se demonstram muito similares àqueles obtidos pelo cálculo teórico como uma função de sua complexidade assintótica, o que confirma a análise deste quesito realizada anteriormente. De maneira análoga, o segundo teste verifica o desempenho do programa em um dos piores casos de execução, caracterizado por todas as palavras serem distintas entre si e a ordenação ser inteiramente realizada por meio do algoritmo Quick Sort. Novamente, os resultados práticos se revelaram muito próximos daqueles obtidos pelo cálculo teórico envolvendo a ordem de complexidade do algoritmo, reafirmando a corretude da análise prévia.

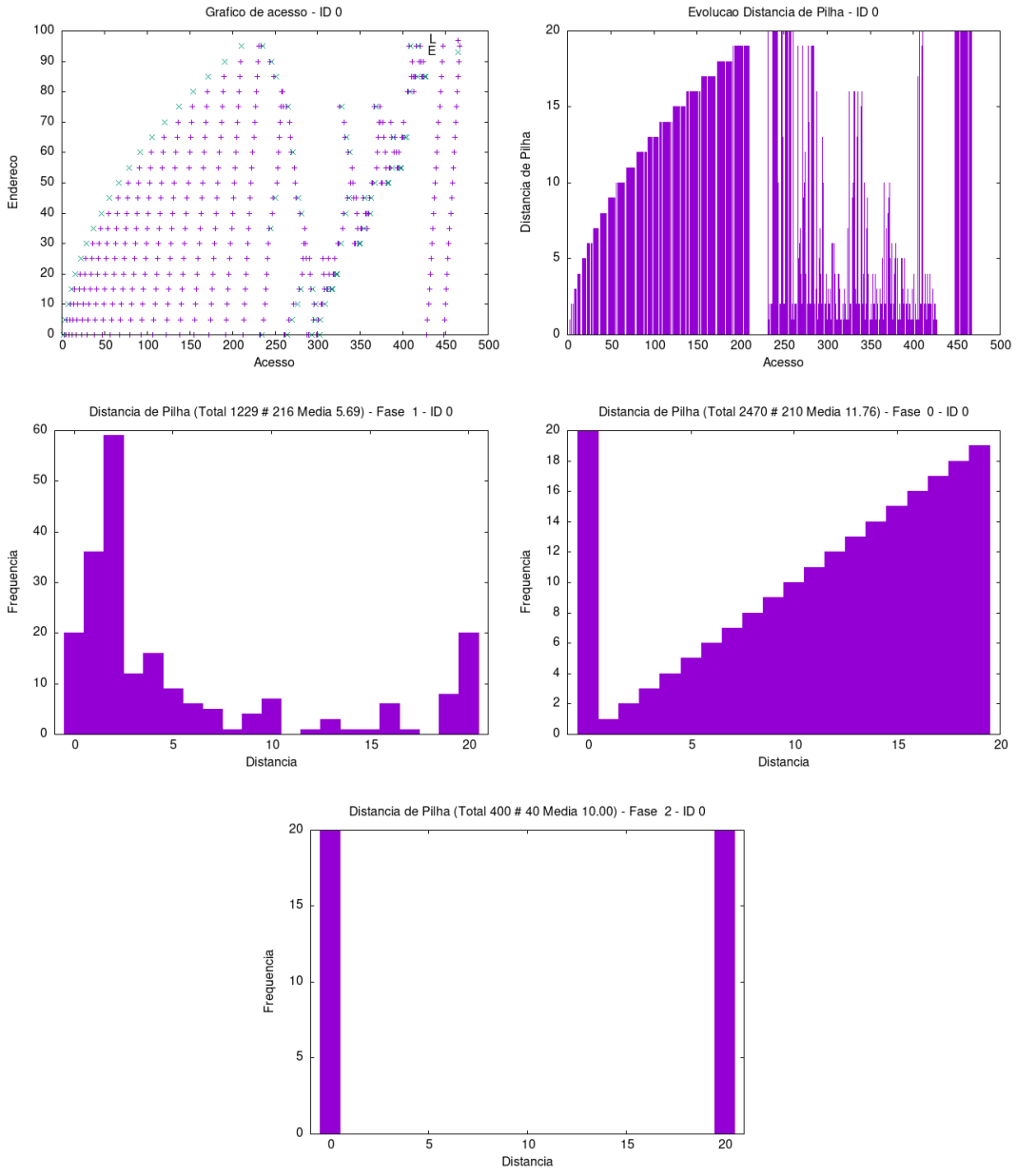
Os terceiro e quarto testes, por sua vez, tinham por objetivo avaliar o impacto de mudanças nos parâmetros de ordenação no tempo de execução do programa para uma mesma situação de operação - mesma entrada de dados: 50 mil palavras, todas distintas entre si. No caso da mediana, o uso de mais elementos da lista em seu cálculo tem por fim minimizar as chances de ocorrência do pior caso de operação do algoritmo Quick Sort. Nos testes realizados, para uma pequena quantidade de elementos, o tempo de execução não varia significativamente, o que apenas indica que, para cada valor, o pivô escolhido foi tão apropriado quanto a escolha do primeiro elemento da lista, sendo este o caso padrão. Entretanto, à medida que mais elementos são considerados, o custo em termos de tempo do cálculo da mediana - sendo esta uma operação O(n^2) - cresce e passa a interferir no desempenho geral da aplicação, deteriorando-o (é possível ver nos resultados que essa situação ocorre a partir de 500 elementos - inclusive). No caso da partição, sua significância diz respeito ao balanço de custo operacional entre os diferentes algoritmos de ordenação. Para valores pequenos de entrada, algoritmos simples, mesmo que apresentem uma ordem de complexidade maior, podem executar mais rapidamente que algoritmos eficientes. Nos testes, essa propriedade é evidente na medida em que, para partições de tamanho razoavelmente pequeno, o uso do Insertion Sort em substituição ao Quick Sort reduziu em alguns centésimos o tempo de execução do programa. Entretanto, de maneira análoga à mediana, quanto mais elementos são considerados a partir de certo ponto, mais o desempenho geral é deteriorado, já que o custo do Insertion Sort cresce de maneira quadrática e passa a superar o custo do Quick Sort, impactando negativamente a aplicação (nos resultados, a partir de 500 elementos - inclusive -, o tempo de execução parou de reduzir e passou a aumentar progressivamente).

b. Localidade de Referência

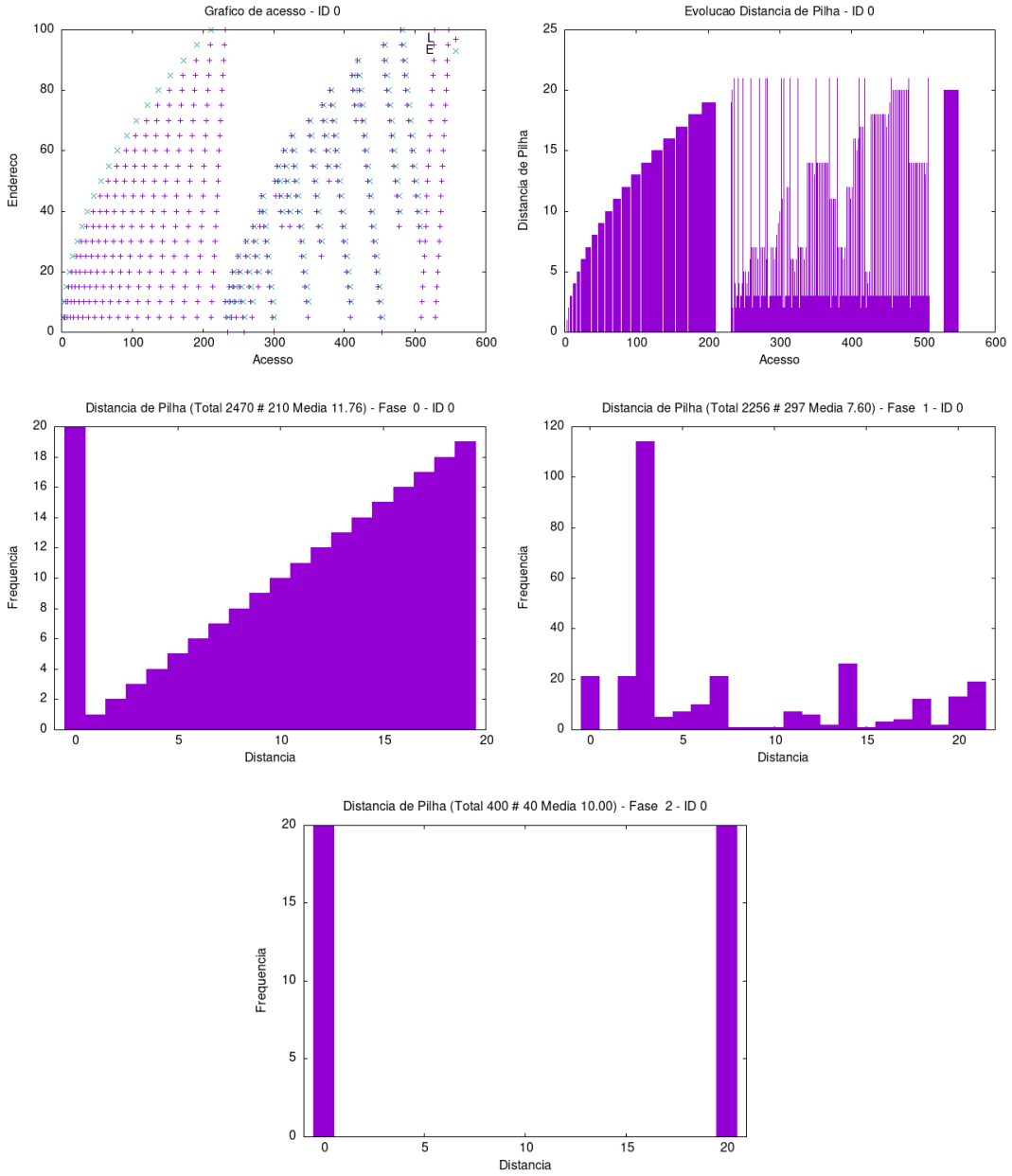
A análise de localidade de referência envolveu testes que verificaram o comportamento dos acessos à memória para diferentes parâmetros de execução do programa, mas em uma mesma condição de operação - mesma entrada de dados, contendo 20 palavras diferentes entre si. Em especial, foi dado destaque à lista de palavras, pois ela seria a principal fonte de manipulações ao decorrer da execução do algoritmo. O registro

desses acessos foi feito com base nas palavras armazenadas em cada célula, o que se justifica por esse elemento ser a referência sobre a qual qualquer operação na lista é feita e, portanto, por seu padrão de acessos expressar a estrutura de dados com um grau satisfatório de precisão. No total, quatro testes foram realizados e, para cada teste, foram definidas três etapas de avaliação, sendo elas: a população da lista, sua ordenação e sua exportação à saída de dados. As três estão juntas no gráfico de acessos, mas separadas nos diagramas de distância de pilha. A seguir, tem-se os resultados obtidos:

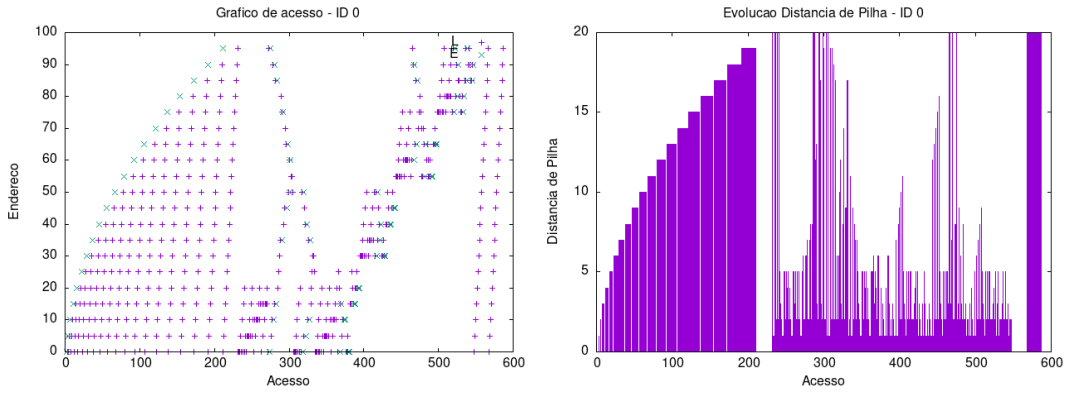
Primeiro Teste: Execução do programa com todos os parâmetros nos valores padrão. O algoritmo Quick Sort é únicamente utilizado para a ordenação e a mediana é sempre o primeiro elemento da partição.

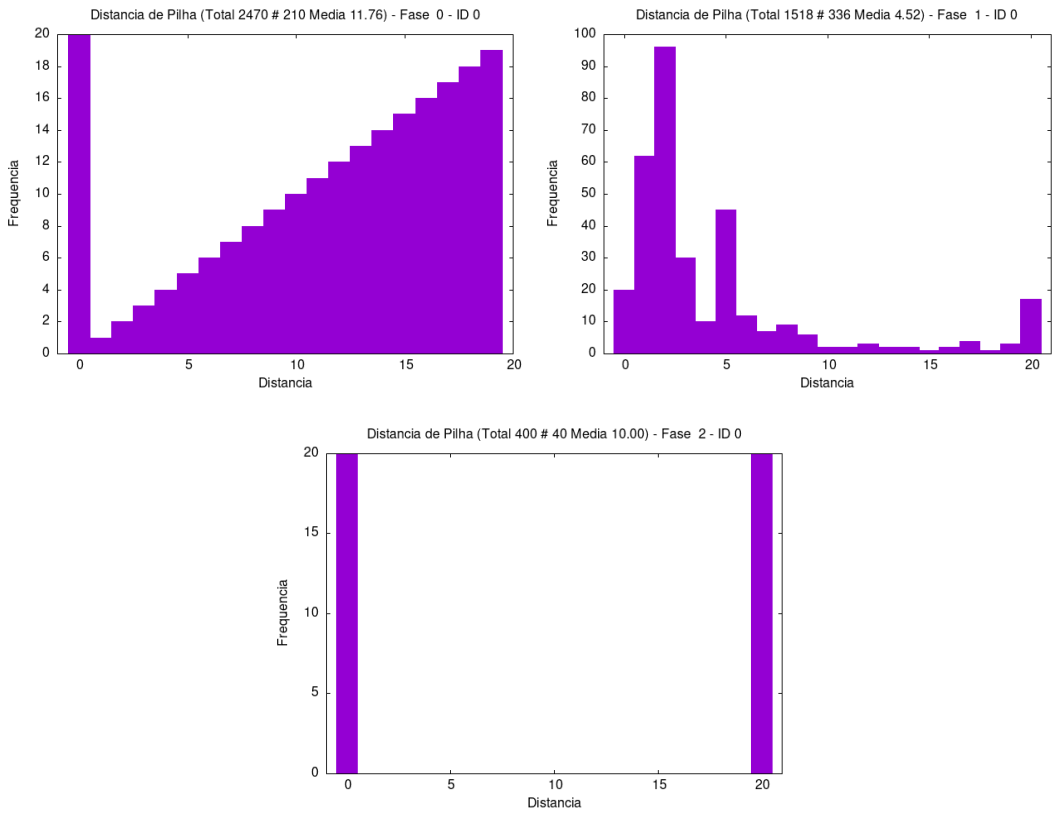


Segundo Teste: Execução do programa com apenas a mediana no valor padrão. O parâmetro de partição foi definido de tal modo que o algoritmo Insertion Sort seja unicamente utilizado para a ordenação.

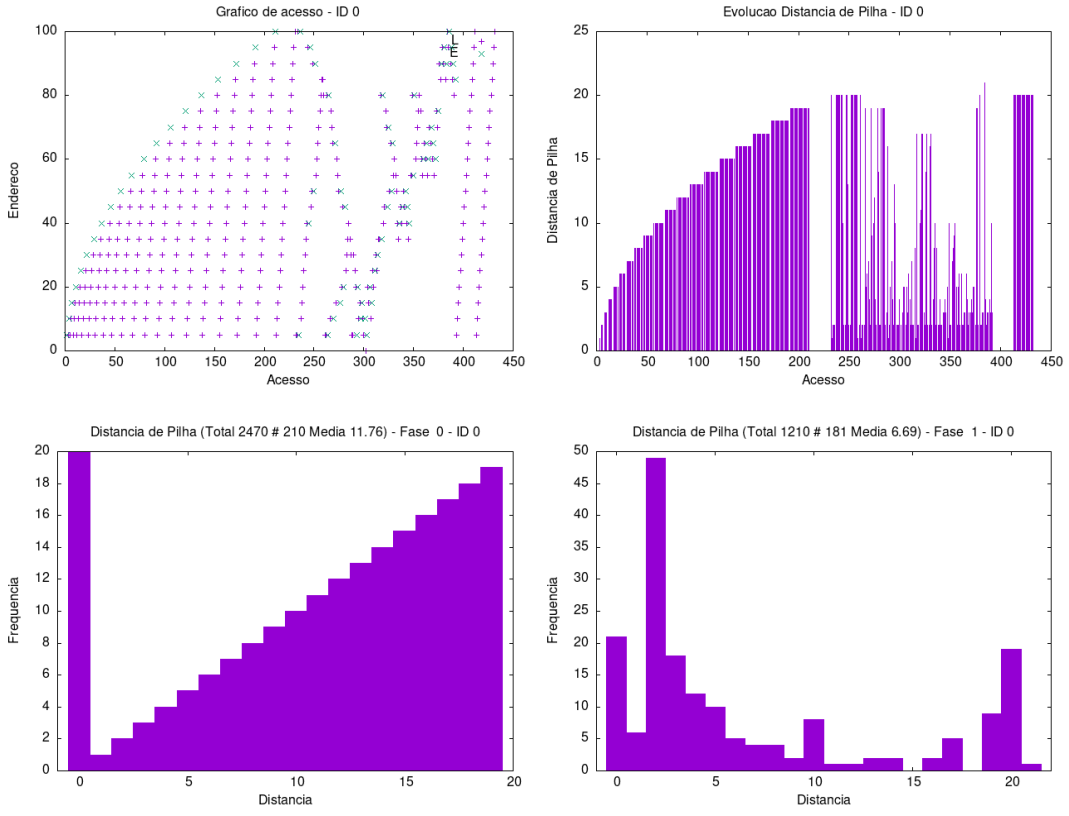


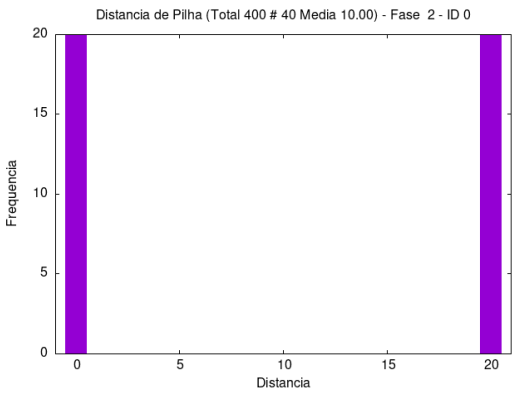
Terceiro Teste: Execução do programa com apenas a partição no valor padrão. O parâmetro de mediana foi definido de modo a considerar 5 elementos. Apenas o algoritmo Quick Sort é utilizado.





Quarto Teste: Execução do programa com apenas a mediana no valor padrão. O parâmetro de partição foi definido de modo que o Insertion Sort seja utilizado para partições com tamanho menor ou igual a 5. A mediana é retornada como o primeiro elemento das partições ordenadas pelo Quick Sort.





Em todos os testes, na fase de população da lista, é nítida sua natureza sequencial de acessos. Como já mencionado, para cada palavra do texto, é inicialmente verificado se ela já está ou não presente na estrutura de dados. Nesse caso - em que todas as palavras são distintas entre si -, isso envolve percorrer todas as células nela já contidas. No gráfico de acessos à memória, esse modo de operação é caracterizado por, inicialmente, cada escrita em uma célula ser precedida de 'i' leituras nas células anteriores. Nos diagramas de distância de pilha correspondentes, esse comportamento é refletido por uma silhueta de "escadaria", já que, para a escrita da célula 'i', todas as 'i-1' células presentes na lista são carregadas em memória para leitura, processo que se repete para as 'n' palavras no texto, originando uma distância de pilha com aspecto "crescente", similar a uma escada.

Na fase de ordenação da lista, cada teste apresenta peculiaridades próprias, as quais são coerentes com os parâmetros utilizados.

No primeiro teste, apenas o algoritmo Quick Sort é executado e não há o cálculo da mediana (retornada como o primeiro elemento da partição). A natureza da operação é pautada na noção de "dividir para conquistar", em que a lista será parcialmente ordenada e depois dividida, iterando-se esse processo. No gráfico de acessos, a ordenação parcial é nítida pela maneira como leituras ocorrem partindo dos extremos opostos da estrutura de dados e se direcionando à posição do pivô, intermediando-se por escritas que indicam a movimentação das células. A divisão, por sua vez, é representada pelo modo com que, ao decorrer dos acessos, a ordenação parcial é iterada e ocorre partindo de extremos cada vez mais próximos entre si, representando as partições menores progressivamente sendo reduzidas.

O terceiro teste apresenta uma silhueta muito similar àquela do primeiro, com a única diferença sendo no cálculo da mediana. Dessa vez, foi estabelecido que esse componente deve ser feito com os 5 primeiros elementos da partição. Tal operação pode ser vista pelos acessos sequenciais que são feitos antes da ordenação propriamente dita, os quais representam o algoritmo percorrendo os primeiros elementos da lista e os comparando entre si para determinar o mediano. Outro detalhe relevante é o fato de a ordenação parcial da primeira partição terminar mais próxima do meio da lista no caso em que há o cálculo da mediana, revelando que seu objetivo de evitar o pior caso do Quick Sort (pivô como um elemento das extremidades) foi sucedido.

No segundo teste, apenas o algoritmo Insertion Sort é utilizado. No gráfico de acessos, novamente, seu modo de atuação é nitidamente visível. Para cada elemento, os anteriores são acessados para leitura de maneira decrescente, enquanto operações de escrita - demarcando a movimentação destes para as células posteriores - são realizadas até que a posição correta do elemento seja encontrada, ponto a partir do qual esse processo se repete iterativamente por toda a extensão da lista.

O quarto teste é peculiar pelo fato de ambos os algoritmos Quick Sort e Insertion Sort serem utilizados, mas seu padrão de acessos mapeia esse comportamento de maneira coerente. Em especial, em

um primeiro momento, o padrão de acessos do Quick Sort é observado. A partir de determinado ponto, entretanto, o padrão de acesso às partições assume a forma do Insertion Sort, demarcando o alcance do número de elementos delimitado pelo parâmetro de troca entre os algoritmos.

No que tange a distância de pilha, os três testes que utilizam predominantemente o Quick Sort apresentam valores bem similares, em contrapartida àquele em que apenas o Insertion Sort é utilizado, para o qual esse valor é significativamente maior, revelando uma menor eficiência deste algoritmo nesse quesito. Entretanto, em todos os casos, o aspecto do diagrama é visualmente disforme, o que é esperado dada a natureza um tanto "abstrata" dos acessos realizados por métodos de ordenação, no sentido de suas operações de leitura e escrita dependerem do formato e da organização da entrada de dados.

Na fase de impressão, apenas acessos sequências de leitura são realizados sobre a lista para a impressão das informações contidas em cada célula. Nesse caso, não há diferença entre cada teste, dado que o conteúdo final em todos será o mesmo - em função de utilizarem a mesma entrada. O gráfico de acessos, assim como o diagrama de distância de pilha, são regulares e conformes com o número de palavras na lista.

6. Conclusões

O presente trabalho prático reafirmou a fundamentalidade de diversos conceitos da área da computação, a saber, **estruturas de dados**, **algoritmos de ordenação** e **análises de desempenho e de localidade de referência**. O diferencial, entretanto, revela-se na aplicação destes a um contexto simples, comum, porém de suma relevância à contemporaneidade, seja ele o **tratamento de textos**, ou, de maneira geral, o **tratamento de informações**. É sob esse pretexto que frui a excepcionalidade da proposta.

Em relação a dificuldades, a ideia do algoritmo em si foi relativamente simples: um texto deveria ser extraído da entrada de dados, cada palavra, percorrida, formatada e contabilizada e, ao final, essas informações deveriam ser exportadas a um arquivo de saída. Um ponto de complicações dentro dessa lógica poderia emergir com relação a detalhes específicos de implementação, isto é, como a linguagem escolhida lida com entradas de texto e quais ferramentas ela oferece para o seu tratamento. Outro ponto de adversidades poderia surgir no que tange a adaptação dos algoritmos e das estruturas de dados para tratarem especificamente com os pormenores da aplicação desenvolvida, sejam eles uma ordenação específica, critérios de corretude ou o domínio das variáveis envolvidas. Porém, a grande e principal dificuldade foi observada no processo de análise computacional. Em particular, os requisitos funcionais definidos abrem uma gama muito vasta de fluxos de execução distintos, cada qual dotado de vantagens, desvantagens, custos operacionais e ordens de complexidade próprios, demandando um estudo mais cuidadoso e abrangente.

As maiores possibilidades, entretanto, vieram justamente das dificuldades encontradas. Em particular, a proposta de se lidar com o tratamento de informações agrega positivamente uma assimilação de conhecimentos e ferramentas úteis para o labor computacional. Além disso, a vastidão em termos de desempenho e de localidade de referência demonstra de maneira prática a maneira com que pequenos ajustes podem apresentar grandes impactos na construção de um programa eficiente, além de que ampliam o escopo de pensamento com relação às possibilidades de elaboração de um algoritmo.

7. Bibliografia

Stack Exchange, Inc.. *Stack Overflow* [Online]. Disponível em: https://stackoverflow.com/ (Acessado em: 03/06/2022).

CPlusPlus. *CPlusPlus* [Online]. Disponível em: https://www.cplusplus.com/ (Acessado em: 03/06/2022).

GeeksforGeeks. *GeeksforGeeks* [Online]. Disponível em: https://www.geeksforgeeks.org/ (Acessado em: 03/06/2022).

8. Instruções para Compilação e Execução

1. Extraia o conteúdo do arquivo .zip disponibilizado.

2. Acesse o diretório **</TP>** por meio do comando **<$ cd TP>**.

3. Execute no terminal o comando **<$ make all>**. Esse comando deverá gerar um executável denominado **tp2** no diretório **</TP/bin>**.

4. A execução do programa é realizada por meio do comando **<$ ./bin/tp2>** seguido dos parâmetros enumerados abaixo, em qualquer ordem:

-i (caminho do arquivo de entrada) : arquivo de entrada de dados para o programa.
-o (caminho do arquivo de saída) : arquivo de destino para a saída do programa.
-m (valor inteiro) : número de valores a serem considerados no cálculo da mediana para a determinação do pivô no algoritmo Quick Sort.
-s (valor inteiro) : tamanho da partição a partir do qual o algoritmo Insertion Sort será utilizado ao invés do Quick Sort.

Um exemplo de execução seria **<$ ./bin/tp2 -i entrada.txt -o saida.txt -m 5 -s 5>**.
Os parâmetros **-i (caminho do arquivo de entrada)** e **-o (caminho do arquivo de saída)** são obrigatórios. Os demais são opcionais e sua ausência irá implicar o uso de valores padrão.

5. O arquivo de saída contendo o resultado do programa para a entrada correspondente deve ser gerado (sobrescrito caso já exista, criado caso ainda não exista) no caminho fornecido.