Trabalho Prático 2
Ordem Lexicográfica

Vitor Emanuel Ferreira Vital

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brazil

vitorvital@dcc.ufmg.br

1. Introdução

Esta documentação tem como objetivo apresentar o decorrer da implementação da resolução do Trabalho Prático 2 da disciplina de Estrutura de Dados. Em tal prática, os principais temas explorados que devem ser solucionados na linguagem C/C++ são:

- Analise arquivos de texto;
- Manipulação de strings;
- Algoritmos de ordenação;
- Desempenho do programa;

A documentação foi dividida em 6 seções principais, incluindo a introdução. A segunda seção trata da descrição das soluções implementada; já na terceira seção é apresentado a análise de complexidade de tempo e espaço dos procedimentos apresentados; a quarta seção é marcada pela descrição, justificativa e implementação dos mecanismos de programação defensiva e tolerância a falhas implementado; na quinta e penúltima seção trabalha-se a análise do desempenho computacional e localidade de referência, assim como as análises dos resultados; por último temos a conclusão geral da documentação.

2. Método

Para resolução do problema proposto o código foi dividido em 7 programas distintos responsáveis por determinadas etapas da resolução. Tais divisões permitem uma maior organização do programa pois cada etapa é responsável por uma função para resolução como um todo. Portanto, foram aglutinadas parte que empenham semelhantes papéis. A configuração dos TAD's foram feitas utilizando as seguintes funções:

- Algoritmo de ordenação simples (Selection Sort);
- Algoritmo de ordenação QuickSort;
- Algoritmo de definição e análise de texto em determinadas ordens lexicográficas;
- Processador de linhas e textos;
- Analisador de desempenho;
- Mecanismos de programação defensiva e tolerância a falhas;
- Lista dinâmica encadeada de palavras;

Precedendo a ordenação textual, o algoritmo preocupa-se, de início, em enfileirar as palavras distintas obtidas do texto, para isso a cada inserção a lista é percorrida verificando se já houve ocorrência da palavra no texto, caso exista não há a adição de uma palavra a lista, apenas é indicado

que existe mais uma ocorrência dessa no texto. Caso não haja ocorrência, essa nova palavra é adicionada ao final da lista.

Em paralelo, tem-se também a preocupação com a definição de uma ordem lexicográfica, utilizando como base a passada no arquivo de inicialização. De início, utiliza-se como padrão a ordem alfabética, caso não exista uma definição explícita na entrada. Entretanto, caso exista foi utilizada a seguinte implementação:

- Definição de um vetor estático de 26 posições (quantidade de letras do alfabeto da língua portuguesa);
- A posição de cada letra respeita a ordem alfabética, ou seja, a primeira posição do vetor é a, a segunda b e assim por diante até a vigésima sexta posição que é da letra z.
- Na fase de definição da nova ordem lexicográfica é realizado o seguinte algoritmo, a primeira letra dessa ordem terá 0 em sua respectiva casa do vetor, ou seja, se z for a primeira, teremos que a vigésima sexta posição do vetor terá valor 0, se d for a segunda a quarta casa será um e assim por diante.

Seguindo tal implementação teremos um vetor de 26 posições no qual cada casa possuirá um número de 0 a 25, não havendo repetições do mesmo. Dessa forma, ao comparar-se duas palavras caso uma letra se distingue de outra verificam-se as posições das letras no vetor e a que possuir o menor número na respectiva casa deve está a frente da que possui maior.

Com isso, é possível ordenar palavras com base nas ordens lexicográficas definidas no início do programa, para isso, tem-se duas possíveis maneiras: ordenar via QuickSort ou Selection Sort.

QuickSort

No algoritmo de ordenação QuickSort divide-se o problema de ordenar um conjunto com n itens em dois problemas menores, os problemas menores são ordenados independentemente dos outros e por fim combinam-se os resultados de forma a produzir a solução final.

De início define-se um pivô, por padrão pega-se o elemento mais à esquerda da partição, caso contrário, o método simples é utilizado para ordenar a partição e assim é obtida a mediana que torna-se o pivô.

Em posse do pivô, percorre-se a lista com dois apontadores, um da direita para esquerda e outro da esquerda para a direita. Caso exista um elemento à esquerda do pivô maior que ele (maior seguindo a novo ordem lexicográfica, ou seja, para o primeiro elemento distinto entre eles o elemento do pivô é menor que o do elemento analisado) o elemento troca de lugar com um elemento menor que o pivô que esteja a direita dele, caso não exista é trocado com o próprio pivô.

Tais operações são feitas até que os apontadores se cruzem na lista, nesse caso temos duas partições, no qual todos os elementos do início ao primeiro vetor encontrado (que vinha da direita para a esquerda) são menores que os elementos a direita da lista (elementos do vetor que vinha da esquerda pra direita até o elemento mais a direita). A partir dessa etapa fazemos a mesma análise até que todo o vetor esteja ordenado.

Porém, como tal algoritmo não é muito eficiente para listas muito pequenas define-se um tamanho de partição máximo, nesse cenário, caso a partição a ser ordenada tenha um número de elementos menor ou igual ao tamanho máximo de partição, utiliza-se o método de ordenação simples, nesse caso o Selection Sort.

Selection Sort

O segundo algoritmo de ordenação utilizado possui implementação mais simples, nessa a lista é ordenada percorrendo ela toda até encontrar o menor elemento da partição e então coloca-se na primeira posição dessa, posteriormente a partição passa a iniciar no elemento seguinte e então o menor elemento dessa próxima partição nele.

Por possuir um alto custo de tempo, não é recomendada sua utilização para listas muito grandes.

Conforme citado anteriormente, a estrutura de dados utilizada foi semelhante a uma fila. Portanto, um elemento novo sempre é inserido ao final dela e ao fazermos a impressão no arquivo imprimimos o primeiro elemento ao último sequencialmente. Entretanto, apesar de, no caso de uma fila convencional, o primeiro elemento inserido é retirado conforme entrou, no programa utilizado é realizada uma ordenação posterior a inserção e só então é realizada uma retirada. Dessa forma, temos um comportamento de inserção e remoção semelhante a de uma fila, porém, por não seguir todos os requisitos é formalmente uma lista dinâmica encadeada, no decorrer da documentação haverá uma explicação mais sucinta do porquê não é formalmente uma fila.

Para toda a análise do programa implementado em linguagem C/C++ (predominantemente C++) e compilado através da ferramenta gdb. Foi utilizado um computador com sistema operacional Linux em um processador Intel® Core™ i5-10210U CPU @ 1.60GHz × 8 com 8Gb de memória RAM.

3. Análise de Complexidade

O algoritmo como um todo possui um custo de tempo e espaço devido às operações realizadas com os elementos da entrada. Portanto, é importante analisar cada função envolvida nessas execuções a fim de verificar se é uma solução ótima para tal caso ou se pode ser substituída por outra função que tornaria a resolução do problema mais eficiente, de forma a balancear os gastos e evitar utilizações de memória e/ou tempo maiores que o realmente necessário.

Função isPoint - complexidade de tempo - essa função verifica se o caractere é um dos sete pontos possíveis, nesse cenário, há a comparação de um caractere com 7 em um loop. Apesar de conter um loop há um número de comparações sempre fixo, independente da entrada, além disso a entrada tem sempre tamanho 1. Portanto, temos uma função com ordem de complexidade constante $\Theta(1)$.

Função isPoint - complexidade de espaço - essa função realiza todas as operações considerando em complexidade assintótica de espaço O(1) além disso o elemento de entrada possui sempre tamanho 1. Assim, a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função lowerTransformer - complexidade de tempo - essa função realiza operações constantes, em tempo O(1). Entretanto, há um laço que percorre toda a string verificando se possui uma letra maiúscula, string essa que pode possuir tamanho n. Dessa forma, a complexidade assintótica de tempo dessa função é $\Theta(n)$.

Função lowerTransformer - complexidade de espaço - essa função realiza todas as operações apenas com uma string passada por referência, portanto o custo de espaço é $\Theta(n)$.

Função getWords - complexidade de tempo - essa função analisa as palavras de um texto e as enfileira, dessa forma temos 3 funções auxiliares. Um texto pode possuir n palavras, portanto, todas elas serão analisadas obtendo um custo O(n), para cada palavra verificamos se os últimos caracteres são pontos e os eliminamos, como podemos ter uma palavra com m letras, sendo as m-1 últimas pontos tem-se para cada uma m*n comparações para ver se é ponto, o que nos dá um custo O(mn), no pior caso e O(n) no melhor, caso não seja um ponto ao final.

Posteriormente cada palavra é colocada no padrão, que no caso é sua versão em minúsculo, para uma palavra com m letras, temos novamente um custo O(mn). Por fim, temos a inserção da palavra já padronizada na lista, como é verificado para cada palavra se houve uma recorrência dela anteriormente para depois inseri-la temos um custo no pior caso de $O(n^2)$, caso não exista, e O(1) caso a lista esteja vazia.

Dessa forma, a complexidade assintótica de tempo dessa função é $\Theta(n^2 + mn)$ no pior caso e $\Theta(mn)$ no melhor.

Função getWords - complexidade de espaço - essa função, não armazena diretamente todas as palavras do texto, entretanto, a função que enfileira sim, logo temos um custo de armazenamento $\Theta(n)$.

Função GetLexOrder - complexidade de tempo - essa função recebe um string de tamanho n, verifica se o caractere é uma letra e cria uma nova string com as letras passadas na ordem utilizada e chama a função que seta a nova ordem. A string pode possuir tamanho de entrada n, logo temos uma complexidade de tempo para sua análise $O(n)$. Já para transformarmos na sua versão em minúsculo teremos apenas 26 letras, sendo essa transformação feita se houver inserção de uma letra maiúscula, logo temos um custo $O(1)$, por fim a definição da nova ordem também é $O(1)$, pois o tamanho da entrada é fixa em 26 sempre. Logo, a complexidade assintótica de tempo dessa função é $\Theta(n)$.

Função GetLexOrder - complexidade de espaço - essa função, armazena indiretamente a string com a nova ordem que pode possuir tamanho n. Logo, a complexidade assintótica de espaço dessa função é $\Theta(n)$.

Função GetTexts - complexidade de tempo - essa função é separada em dois casos, caso a parte a ser analisada seja um texto e o outro caso seja a ordem lexicográfica. Sendo uma nova ordem, chama-se a funcao GetLexOrder que tem complexidade de tempo $O(n)$, caso contrário chamamos a função getWords que possui complexidade de tempo $\Theta(n^2 + mn)$ no pior caso e $\Theta(mn)$ no melhor, nesse segundo caso chamamos-a para cada linha do texto, portanto podemos ter k linhas no texto. Assim, a complexidade assintótica de tempo dessa função é $\Theta(kn^2+kmn)$.

Função GetTexts - complexidade de espaço - essa função armazena diretamente os dados de GetLexOrder e de getWords, logo sua complexidade de espaço é $\Theta(n)$, no qual n é a quantidade de palavras do texto.

Função Order - complexidade de tempo - essa função realiza operações constantes de tempo $O(1)$. Além disso, há um laço com número fixo de repetições iguais a 26, logo, a complexidade assintótica de tempo dessa função é $\Theta(1)$.

Função Order - complexidade de espaço - essa função inicializa um vetor com 26 elementos atribuindo um valor de 0 a 25 a cada um deles. Assim, a complexidade assintótica de espaço dessa função é $O(1)$.

Função newOrder - complexidade de tempo -essa função realiza operações constantes de tempo $O(1)$. Além disso, há um laço com número fixo de repetições iguais a 26, logo, a complexidade assintótica de tempo dessa função é $\Theta(1)$.

Função newOrder - complexidade de espaço -essa função atribui valores de 0 a 25 a um vetor com 26 elementos. Assim, a complexidade assintótica de espaço dessa função é $O(1)$.

Função convert - complexidade de tempo - essa função realiza uma operação constante, logo a complexidade assintótica de tempo dessa função é $\Theta(1)$.

Função convert - complexidade de espaço -essa função realiza uma operação constante convertendo um caractere de ASCII para nova ordem lexicográfica estabelecida, logo a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função strcmp - complexidade de tempo - essa função compara duas strings realizando operações contínuas em um loop que percorre os elementos das strings paralelamente, logo temos custo $O(n)$ no pior caso e $O(1)$ no melhor caso, que é quando a primeira letra dos dois se diferem. Essa função chama a convert 2 vezes, sendo os dois custos $O(1)$. Assim, a complexidade assintótica de tempo dessa função é $\Theta(n)$ no pior caso e $\Theta(1)$ no melhor.

Função strcmp - complexidade de espaço - essa função armazena indiretamente 2 strings de tamanho n e m, logo a ordem de complexidade de espaço dessa função é $\Theta(n+m)$.

Função QuickSort - complexidade de tempo - essa função chama a OrderList. Portanto, a complexidade assintótica de tempo dessa função é $\Theta(nlogn)$ em média, e $O(n^2)$ no pior caso.

Função QuickSort - complexidade de espaço - essa função apenas manipula uma lista e todas variáveis utilizadas são constantes, dependendo apenas do número de chamadas recursivas, logo, no caso médio temos complexidade assintótica de espaço dessa função igual a $\Theta(nlogn)$

Função OrderList - complexidade de tempo - essa função chama a responde a ordenação via QuickSort. Portanto, a complexidade assintótica de tempo dessa função é $\Theta(nlogn)$ em média, e $O(n^2)$ no pior caso.

Função OrderList - complexidade de espaço - essa função é recursiva logo temos o custo da de todas essas chamadas, resultando em uma complexidade assintótica de espaço igual a $\Theta(nlogn)$.

Função getPivot - complexidade de tempo - essa função, no melhor caso apenas seta o elemento mais à esquerda como pivô, no pior caso, chama-se a função que ordena via selection sort, que possui custo $O(n^2)$ e posteriormente caminha até o meio da lista, o que gera mais n/2 iterações, como resultado temos a complexidade assintótica de tempo dessa função igual a $\Theta(n^2)$ no pior caso e $\Theta(1)$ no melhor.

Função getPivot - complexidade de espaço - essa função realiza todas as operações considerando estruturas auxiliares unitárias O(1)Assim, a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função Partition - complexidade de tempo - essa função realiza uma chamada da função get pivô, que possui complexidade assintótica $\Theta(n^2)$ no pior caso e $\Theta(1)$ no melhor, além disso, realiza a chama da função strcmp n vezes que possui custo $\Theta(n)$ no pior caso e $\Theta(1)$ no melhor, obtendo assim um custo, pelas n chamadas, $\Theta(n^2)$ ou $\Theta(n)$, por fim a função swap é chamada n vezes e possui custo $\Theta(1)$, gerando um custo $\Theta(n)$. Portanto, a complexidade assintótica de tempo da função partition é $\Theta(n^2)$ no pior caso e $\Theta(n)$ no melhor

Função Partition - complexidade de espaço - essa função realiza todas as operações, considerando estruturas auxiliares, com custo de espaço O(1). Assim, a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função SelectionSort - complexidade de tempo - essa função é o construtor da classe e realiza a chamada da função Orderlist apenas, por tanto sua complexidade de tempo é diretamente ligada a complexidade da função OrderList que possui complexidade assintótica de tempo $\Theta(n^2)$;

Função SelectionSort - complexidade de espaço - essa função realiza uma chamada da função, além disso a função chamada também gera um custo constante de espaço. Assim, a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função OrderList - complexidade de tempo - essa função é a implementação do algoritmo de ordenação selection sort, nessa temos uma complexidade assintótica de tempo igual a $\Theta(n^2)$. Além disso, temos a chamada da função strcmp dentro da função, sendo chamada n^2 vezes, como o custo dela depende do tamanho da string, tendo custo $\Theta(m)$, temos então, para função orderlist uma complexidade de tempo igual a $\Theta(n^2m)$.

Função OrderList - complexidade de espaço - essa função realiza todas as operações, considerando as estruturas auxiliares com custo unitário. Portanto, a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função Word - complexidade de tempo - essa função é a construtora da classe Word e cria uma nova estrutura, em um custo unitário. Logo, a complexidade assintótica de tempo dessa função é $\Theta(1)$.

Função Word - complexidade de espaço - essa função cria apenas uma estrutura, sendo a entrada constante, logo temos uma complexidade assintótica de espaço $\Theta(1)$ para essa função.

Função ListWords - complexidade de tempo - essa função é a construtora da classe ListWords e é responsável por inicializar os apontadores da lista. Portanto, a complexidade assintótica de tempo dessa função é $\Theta(1)$.

Função ListWords - complexidade de espaço - assim como a construtora da classe Word, essa classe apenas cria e inicializa os apontadores da lista. Dessa forma , a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função ~ListWords - complexidade de tempo - essa função é a destrutora da classe ListWords, para desalocar as células ela deve percorrer a lista desalocando cada uma das células. Nesse caso, tem-se uma complexidade assintótica de tempo igual a $\Theta(n)$.

Função ~ListWords - complexidade de espaço - essa função desaloca a memória e possui custo e possui estruturas auxiliares constantes. Portanto, a complexidade assintótica de espaço é $\Theta(1)$.

Função AddNewWord - complexidade de tempo - essa função adiciona um novo elemento ao final da lista, entretanto verifica se a existência da palavra nessa fila, por isso, para cada inserção há uma verificação de n elementos. Nesse caso, tem-se uma complexidade assintótica de tempo igual a $\Theta(n)$ no pior caso e $\Theta(1)$ no melhor, ou seja, quando a lista está vazia.

Função AddNewWord - complexidade de espaço - essa função adiciona uma nova célula ao final da lista, além disso as estruturas auxiliares possuem custo unitário. Portanto, a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função swap - complexidade de tempo - essa função troca os dados de duas células. Portanto, a complexidade assintótica de tempo dessa função é $\Theta(1)$.

Função swap - complexidade de espaço - essa função apenas utiliza estruturas auxiliares que possuem custo unitário. Sendo assim, a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função ListAccess - complexidade de tempo - essa função percorre a lista a fim de fazer um registro de memória. Logo, a complexidade assintótica de tempo dessa função é $\Theta(n)$.

Função ListAccess - complexidade de espaço - essa função apenas utiliza estruturas auxiliares que possuem custo unitário. Sendo assim, a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função Print - complexidade de tempo - essa é responsável por escrever todos os dados da lista no arquivo. Portanto, a complexidade assintótica de tempo dessa função é $\Theta(nm)$, no qual n é a quantidade de palavras e m a quantidade de caractere por palavra.

Função Print - complexidade de espaço - essa função apenas utiliza estruturas auxiliares que possuem custo unitário. Sendo assim, a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Programa total - complexidade de tempo - o custo de tempo do programa depende diretamente da complexidade assintótica das funções utilizadas por ele. Portanto, como a função mais custosa dentre as chamadas na execução é a getTexts a complexidade assintótica de tempo do programa é $\Theta(kmn+kn^2)$.

Programa total - complexidade de espaço - como citado, o programa é responsável por armazenar todos as palavras do arquivo em uma lista, podendo possuir n palavra com m caracteres cada. Portanto, o custo assintótico de espaço do programa como um todo é $\Theta(mn)$.

4. Estratégias de Robustez

Os mecanismos de programação defensiva e tolerância a falhas são de extrema importância na realização de programas, isso porque evita que o usuário conceda entradas incorretas, utilize dessas falhas para danificar de alguma forma o programa, ou obtenha resultados inesperados na execução das funções do programa.

Nessa etapa foram utilizadas as funções __erroassert e __avisoassert para informar ao usuário problemas durante a execução do programa ou por introdução de valores incorretos.

Nome do arquivo de entrada

Ao inicializar o programa é necessário passar o arquivo no qual os dados utilizados pelo programa estão, dessa forma, é necessário verificar se existe a passagem do nome do arquivo para sua futura abertura.

Abertura dos arquivos de entrada e saída

Na implementação desse programa é necessário obter dados de um arquivo de entrada, além de gravar os resultados obtidos em um arquivo de saída, ou seja, a utilização de arquivos é de extrema importância no decorrer do programa. Entretanto, podem haver falhas na abertura desses, o que impossibilita a execução do programa. Portanto, é de suma importância verificar se os arquivos foram abertos corretamente para utilização no programa.

Lista de palavras

O programa tem como objetivo obter as palavras do texto ordenando-as e verificando o número de ocorrências. Dessa forma, antes da ordenação da lista é necessário verificar se existem dados presentes na lista, caso contrário houve falha na atribuição de palavras ou na utilização do programa.

Fechamento dos arquivos

Durante o programa é fundamental a utilização de arquivos, porém ao findar da execução é necessário que os arquivos sejam fechados. Entretanto, a falha no fechamento não é fundamental para o prosseguimento do programa, mas algo inesperado. Portanto, é necessário alertar o usuário quanto a situação, porém não há necessidade de encerrar o programa como um todo.

Referência aos extremos

Durante a ordenação via QuickSort é necessário a definição de um pivô que estará entre um elemento mais à esquerda até um elemento mais à direita. Como o sentido de acesso a lista é da esquerda para a direita é necessário que a referência do valor à esquerda esteja mais próximo do início da lista que o da direita.

Quantidade de valores para mediana

O programa tem como objetivo ordenar uma lista de palavras, caso utilize-se do método de ordenação QuickSort é interessante a definição de um pivô cujo valor não seja o menor ou o maior da lista, para isso são calculadas medianas de partições da lista. Assim, ao definir o tamanho da partição é necessário que ela possua menos elementos que a lista como um todo, porém caso não haja é necessário alertar o usuário da situação, evitando casos inesperados, e setar esse valor para um caso padrão válido.

Troca de dados das células

Durante a troca de elementos entre duas células é necessário que a referência a elas sejam válidas e realmente indiquem um elemento da lista. Dessa forma, caso um dos elementos não seja válido é necessário alertar o usuário, não realizando a troca entre eles para não haver perda dos dados.

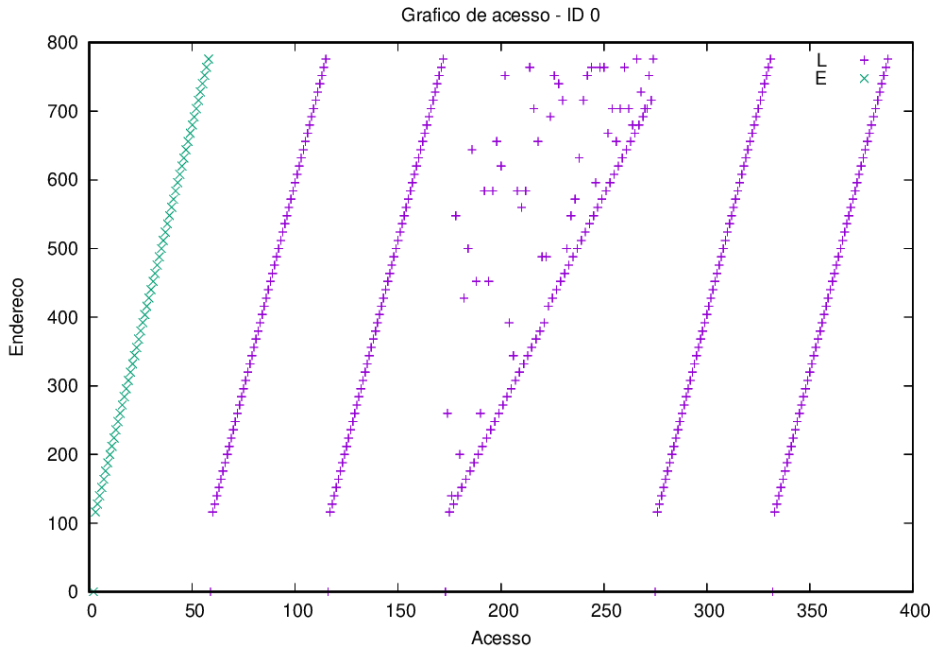
5. Análise Experimental

O programa pode ser dividido em 2 análises principais, sendo estas relativas ao algoritmo de ordenação utilizado. Com base nisso, será feita a análise de desempenho computacional e localidade de referência dos experimentos realizados.

SelectionSort

O algoritmo de ordenação selection sort possui um custo quadrático no processo de ordenação. Além disso, durante sua inserção todos os elementos do vetor são acessados até obter-se o menor valor e isso é realizado até que o vetor esteja ordenado.

SelectionSort - Mapa de acesso

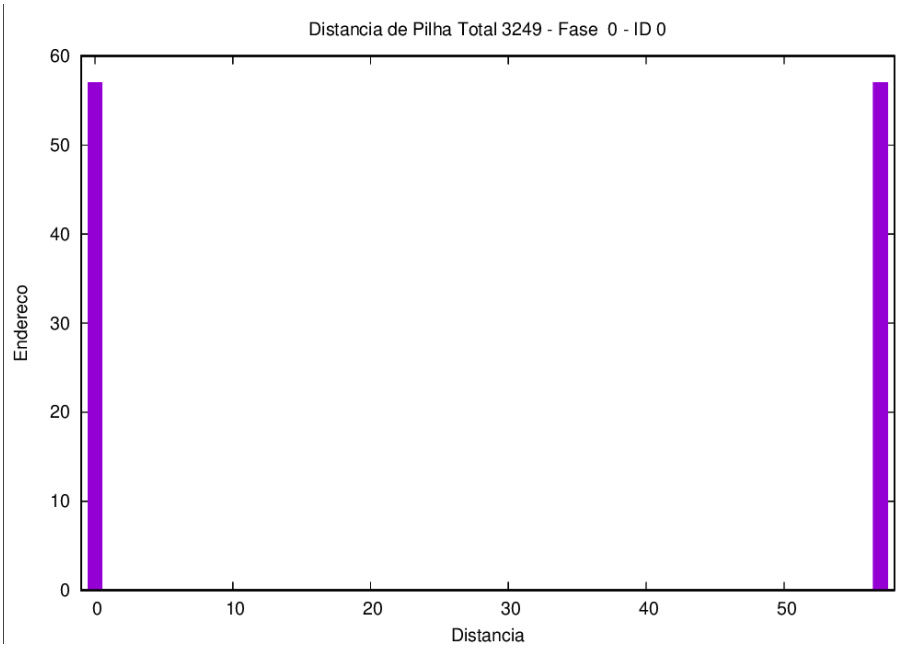


Podemos ver no mapa de acesso a ordenação de uma lista com aproximadamente 60 elementos. Nesse gráfico, inicialmente temos a inicialização da lista com os elementos passados, posteriormente temos dois acessos a ela, sem alteração dos seus valores. Já, na quarta etapa, temos um acesso com partes sequenciais, porém mesmo assim alguns elementos que não estão em endereços seguidos. Isso ocorre pois a lista inicialmente não está ordenada, dessa forma elementos que estão em endereços distantes são acessados, pois os valores contidos nele não estão corretos, nessa etapa ocorre o acesso a essa célula para obter-se o valor dela, posicionando-a em sua devida célula e o valor que estava ocupando é transferida para célula que ficou vazia.

Posteriormente, temos mais 2 novos acessos de leitura uma para mapear a distância de pilha e a segunda referente a impressão da lista no arquivo de saída.

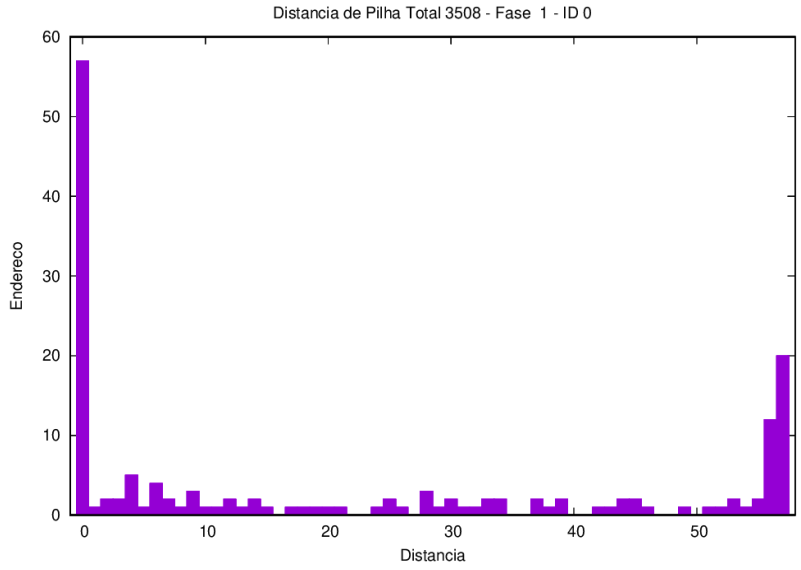
Como citado, temos três etapas distintas ao executarmos o programa: a primeira de inicialização, a segunda de ordenação e por último a impressão. Com base nisso, pode-se realizar a análise das distâncias de pilha.

SelectionSort - Inicialização



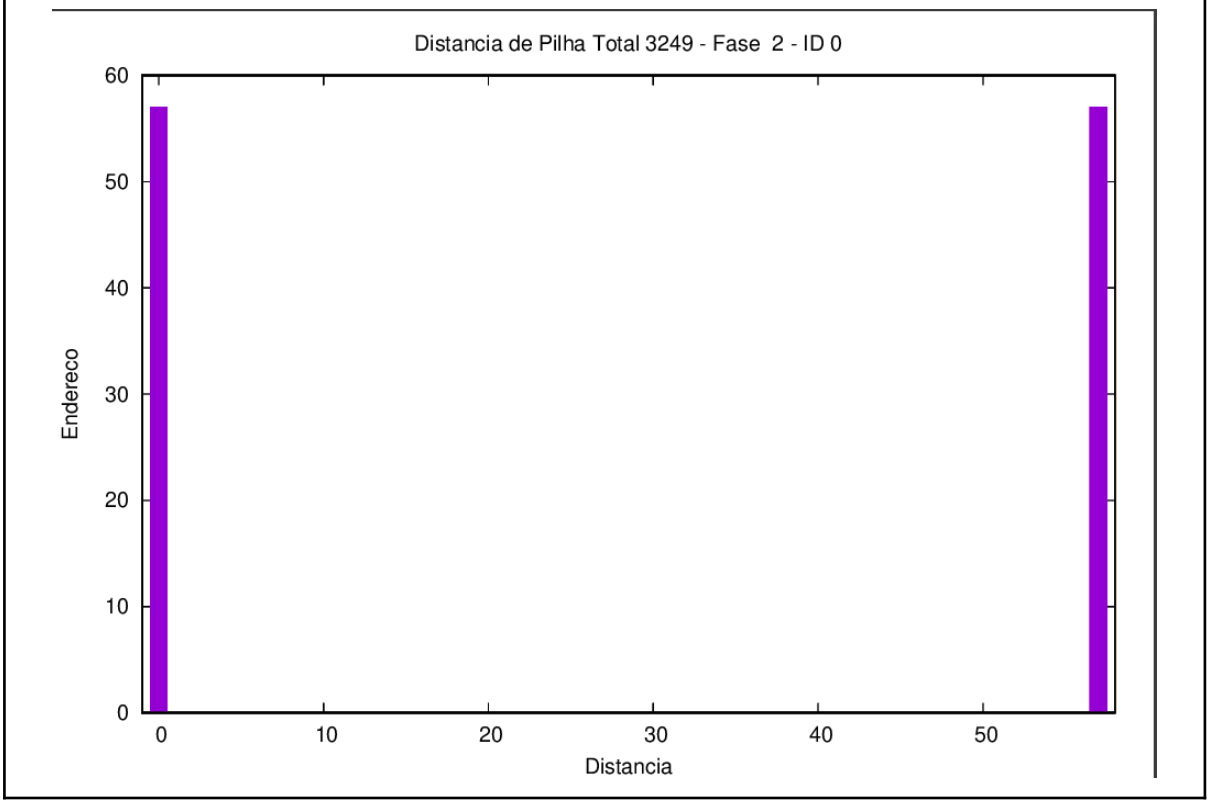
Nessa etapa, por haver apenas a inicialização e o acesso, temos inicialmente todas as distâncias de pilha igual a 0 e posteriormente o primeiro elemento a ser inserido é, também o primeiro a ser acessado, por ser uma pilha esse elemento se encontra a maior distância do topo, tem-se, por isso, 57 acessos à distância de pilha 0 e posteriormente 57 acessos a distância de pilha 57.

SelectionSort - Ordenação



Na etapa da ordenação, após o acesso inicial, justificando os 57 acessos a distância 0, tem-se distâncias não padrões sendo acessadas, isso ocorre pois, a ordem inicial dos elementos é aleatória, portanto, ao buscar o menor elemento da partição ele não está em uma posição específica. Além disso, células contém valores que não necessariamente deveriam estar nelas e, por isso, são trocadas diversas vezes, gerando um histograma não padronizado como é o caso da etapa de inicialização.

SelectionSort - Impressão



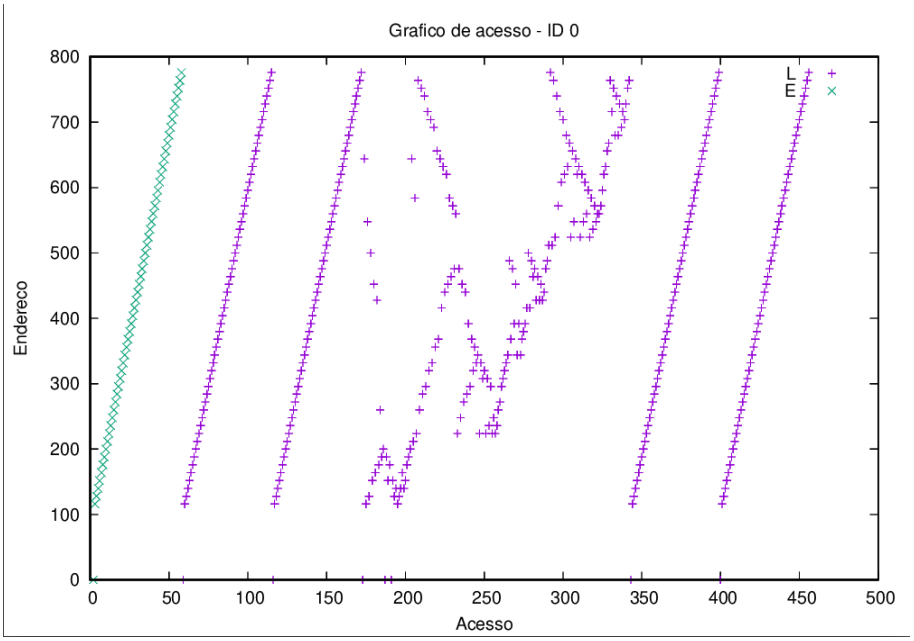
Por fim, a etapa de impressão, nessa os elementos são inicialmente acessados para permitir empilhar-los, em seguida os elementos da lista são acessados da esquerda para a direita, ou seja, o primeiro elemento a ser empilhado é também o primeiro a ser acessado, por estar a maior distância do topo da pilha acumula-se acessos à distância 57. Comportamento semelhante ao da inicialização.

QuickSort

O algoritmo de ordenação quicksort possui um custo nlogn no processo de ordenação. Além disso, durante sua inserção todos os elementos do vetor são inicialmente posicionados de forma a ficar na direita caso seja maior que o pivô e a esquerda caso seja menor que o pivô. Posteriormente a partição direita e esquerda são analisadas e ordenadas independente uma da outra.

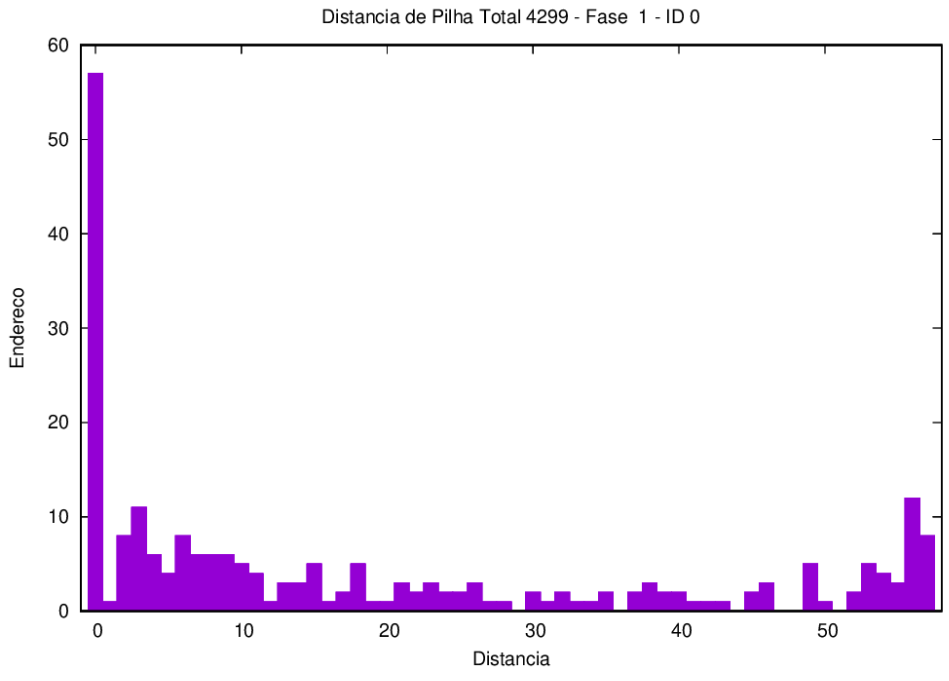
Assim como no algoritmo selection sort utilizado anteriormente o arquivo inicial possui 57 elementos, dessa forma, como o processo de inicialização e impressão independem da forma como a lista será ordenada elas não serão replicadas na etapa do quicksort, pois seguem o mesmo comportamento.

QuickSort - Mapa de acesso



O mapa de acesso do algoritmo de ordenação quicksort inicialmente possui a inicialização da lista, posteriormente são realizados 2 acessos de leitura a lista, sem alteração dos valores. Na quarta etapa, tem-se de fato o algoritmo funcionando. Pode-se reparar que os elementos são permutados diversas vezes dentro do algoritmo, isso ocorre porque, diferente do selection sort, as permutações não ocorrem para posicionar os dados no seu devido local e sim posicioná-los em sua partição. No decorrer do algoritmo a partição torna-se unitária e a localização no qual o dado está coincide com sua devida posição. Por conta disso, temos tal padrão de acesso.

QuickSort - Ordenação

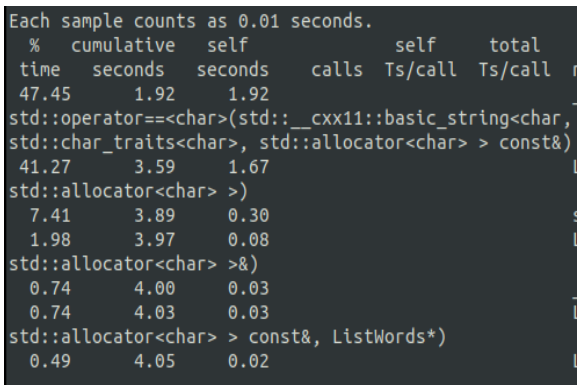


Da mesma forma que ocorre com o histograma da ordenação do selection sort, no quicksort não há, de início, uma ordem correta de onde os dados vão estar. Dessa forma, a permutação entre elementos ocorre de maneira não padronizada, isso porque não há uma posição correta de onde o elemento maior ou menor que o pivô estarão, gerando acessos a elementos que estão a distâncias de pilha não bem definidas.

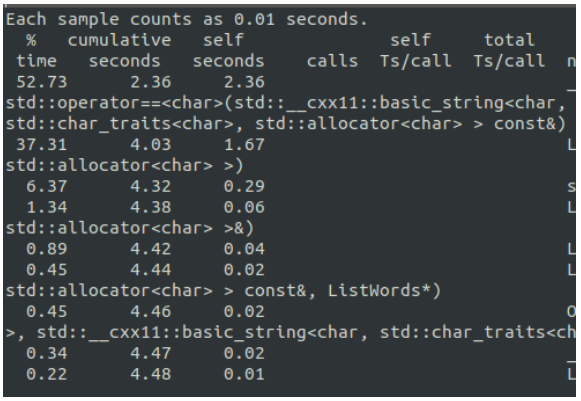
Tempo de execução

Comparando as execuções dos dois métodos para arquivos com grandes quantidades de texto, podemos ver que o desempenho do quicksort foi superior ao do selection sort. Podemos visualizar dois casos abaixo, para um arquivo com 40 mil linhas e outro com 120 mil linhas.

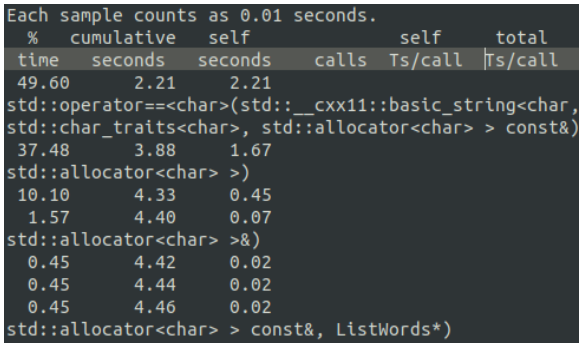
QuickSort - 40 mil



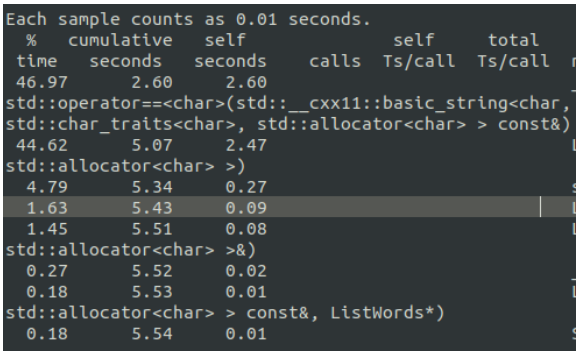
Selection Sort - 40 mil



QuickSort - 120 mil



Selection Sort - 120 mil



Com base nisso, podemos ver que em geral o quicksort é mais eficiente que o Selection Sort para listas muito grandes.

6. Conclusão

Este trabalho lidou com o problema de algoritmos de ordenação, com tal intenção foi necessário ordenar um texto com base em uma nova ordem lexicográfica. Assim, a abordagem utilizada para ordenar as palavras contidas no texto foram o QuickSort e o Selection Sort.

Com a solução foi possível verificar a eficiência de cada um dos métodos de forma a utilizá-los quando for mais conveniente e útil. Além disso, foi de extrema importância verificar formas de tornar métodos já existentes ainda mais eficientes, seja usando o pivô, no caso do quicksort; seja ordenar partições menores via algoritmos simples. Dentre outras ferramentas que agregam e visão analítica de problemas e métodos já existentes.

Por meio da resolução desse trabalho, foi possível praticar conceitos relacionados a algoritmos de ordenação, estruturas de dados, resoluções de problemas e análise de complexidade. Além disso, permitiu diferenciar ainda mais a perspectiva de apenas programar para resolver problemas, para programar com o objetivo de encontrar a solução mais eficaz para os problemas, entendendo a implementação e seu comportamento na memória e em custo de tempo.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo a utilização do algoritmo recursivo do quicksort unida as listas dinâmicas, principalmente com a ligado a utilização dos ponteiros. Entretanto, com o decorrer do processo e insistência na sua resolução foi possível agregar muito conhecimento no âmbito de programação de computadores.

7. Referências

www.stackoverflow.com

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Pascal e C

Cormen, T. Leiserson, C, Rivest, R., Stein, C.(2009) Introduction to Algorithms

Aulas e materiais disponibilizados no moodle UFMG.

8. Instruções para compilação e execução

Na pasta TP há um arquivo makefile responsável por compilar os testes e análises de desempenho e localidade do programa. Basta que, no terminal, seja digitado make para rodar todos os testes.

Por padrão o main.cpp abre o arquivo in.txt na pasta TP para executar o programa e sua saída estará na mesma pasta no arquivo out.txt, entretanto, todos esses dados podem ser alterados na chamada do programa. Além das opções definidas na descrição do TP, no qual define-se o número de partições máxima, o nome do arquivo de entrada e saída, etc, foi adicionado também mais duas opções a -[p|P] e a -[l|L], nessas são possíveis definir, respectivamente, o endereço de saída do registro de desempenho e se a opção de padrão de acesso e localidade será ativada, para ativá-la basta que a opção -[l|L] seja descrita ao executar o programa.

Para o padrão de acesso e localidade a pasta analisamem disponibilizada pelo Wagner/Gisele no moodle deve estar na mesma pasta que contém a pasta TP .

Por padrão, no Makefile, todos os arquivos de saída são gerados em pastas separadas na /tmp/.

A chamada da função pode ser feita da mesma forma que foi definida na especificação do TP2 disponibilizada no moodle. Porém, tem-se por padrão do programa:

- Arquivo de entrada - “/TP/in.txt”
- Arquivo de saída - “/TP/out.txt”
- Quantidade de valores para cálculo da média - 1
- Quantidade máxima da partição para utilização de algoritmo simples 1
- Arquivo de saída de análise do registro de desempenho - “”
- Opção de padrão de acesso e localidade - false