Trabalho Prático 1 – Múltiplas Ordenações

João Pedro Figueiredo Bicalho
2023110747

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

jpedrofb@ufmg.br

1. Introdução

Esta implementação tem como objetivo ordenar indiretamente uma base de dados xCSV lida de um arquivo. Serão utilizadas 3 chaves de ordenação diferentes: nome, CPF e endereço, e então serão imprimidas as listas ordenadas por cada chave.

Para resolver este problema, foi seguida uma abordagem de implementar um array de índices com o mesmo número de elementos da lista que deve ser ordenada e ordenando esse array em seu lugar, diminuindo o custo de mover os dados contidos na lista.

2. Método

O programa foi desenvolvido na linguagem C++, e compilado pelo compilador G++. O sistema operacional utilizado é WSL - Ubuntu. O processador utilizado é um Intel Core i5-1135G7 @ 2.40GHz (8 CPUs), ~2.4GHz, e 16GB de memória RAM.

Está organizado na seguinte estrutura: *OrdenacaoIndireta.hpp* contém definições de valores constantes, da estrutura de dados que será utilizada de nome *OrdInd*, e das funções que serão utilizadas. *OrdenacaoIndireta.cpp* contém a implementação destas funções, e *main.cpp* chama essas funções e as utiliza conforme requisitado.

É utilizado um array de ponteiros para a struct *OrdInd*, facilitando o acesso por índice e a ordenação do array de índices.

As funções *Cria* e *CarregaArquivo* lêem o arquivo e criam um array de ponteiros *OrdInd,* lendo e armazenando os dados de cada pessoa dentro da estrutura. Um array de índices de tamanho igual ao número de pessoas na base de dados também é criado, o qual será usado para realizar a ordenação indireta.

Após isso o código utiliza de três algoritmos de ordenação – QuickSort, SelectionSort e Shellsort – para ordenar o array de índices. Cada algoritmo é utilizado três vezes, uma para cada chave.

3. Análise de Complexidade

Começaremos a análise de complexidade a partir das funções *Cria* e *CarregaArquivo*.

Função Cria – complexidade de tempo: essa função é responsável por ler as linhas iniciais do arquivo e obter o número de pessoas que serão inseridas na struct, alocando a memória necessária. Essas ações têm complexidade de tempo $O(1)$ e $O(n)$ respectivamente, onde n é o número de pessoas cujo dados serão inseridos na estrutura, levando a uma complexidade de tempo total de $O(1)$ + $O(n)$ = $O(n)$.
Função Cria – complexidade de espaço: essa função aloca o espaço necessário de acordo com um número de pessoas n para armazenar uma quantidade de dados por pessoa que chamaremos de m, logo possui complexidade de espaço $O(n \cdot m)$.

Função CarregaArquivo – complexidade de tempo: Essa função itera por toda a base de dados com n registros, inserindo os dados no array de ponteiros. O campo payload de cada registro tem um tamanho diferente dependendo do arquivo de entrada, portanto levaremos o tamanho da carga em conta na análise de complexidade o chamando de m. A função insere n vezes cada registro dominado pelo tamanho m, logo sua complexidade assintótica de tempo é $O(n \cdot m)$.
Função CarregaArquivo – complexidade de espaço: Essa função recebe os arrays em que irá atuar via parâmetro, portanto possui complexidade de espaço $O(1)$.

Função shellSort – complexidade de tempo: O algoritmo Shellsort utilizado foi de sequência $n/2^{\mathrm{i}}$, o qual possui complexidade de tempo $O(n^2)$.

Função shellSort – complexidade de espaço: O algoritmo não aloca memória, apenas recebendo os arrays via parâmetro, portanto possui complexidade de espaço $O(1)$.

Função selectionSort – complexidade de tempo: A complexidade de tempo do algoritmo Selectionsort é $O(n^2)$, pois possui dois for aninhados que iteram por toda a base de dados n.

Função selectionSort – complexidade de espaço: O algoritmo não aloca memória, apenas recebendo os arrays via parâmetro, portanto possui complexidade de espaço $O(1)$.

Função quickSort & partição – complexidade de tempo: Analisaremos o Quicksort e partição em conjunto, pois a função partição está completamente envolvida pelo Quicksort. O Quicksort possui uma complexidade assintótica de tempo média de $O(n\ log\ n)$, ou $O(n^2)$ no pior caso.

Função quickSort & partição – complexidade de espaço: O Quicksort como algoritmo utiliza uma pequena pilha como memória auxiliar devido à sua recursividade, possuindo complexidade de espaço $O(log\ n)$ no caso médio e $O(n)$ no pior caso.

O arquivo *main*, responsável por utilizar as funções conforme requisitado pelo enunciado, possui três loops que percorrem o array de ponteiros que contém a base de dados e os imprime, e portanto possui complexidade de tempo $O(n)$. Todavia *main* também aloca o vetor de índices que será utilizado na ordenação indireta, fazendo com que sua complexidade de espaço seja $O(n)$.

Isso traz a complexidade de tempo total do algoritmo para:

$$O(n \cdot m) + O(n) + O(n^2) + O(n^2) + O(n \log n) = O(n \cdot m)$$

E a complexidade de espaço total do algoritmo é

$$O(n \cdot m) + O(\log n) + O(n) = O(n \cdot m)$$

4. **Estratégias de robustez**

Tanto o array de ponteiros que armazenará os dados quanto o array de índices utilizado para a ordenação indireta possuem tamanhos lidos a partir do arquivo que é recebido, portanto evitando problemas de transbordamento de dados.

Além disso, verificamos o argumento passado como nome do arquivo afim de verificar se o mesmo é nulo, encerrando a execução e enviando uma mensagem de erro caso seja nulo.

5. **Análise Experimental**

Começaremos a análise experimental através de uma avaliação do desempenho do programa, medindo o tempo que leva para a execução das funções mais caras. Para este fim, utilizaremos os arquivos disponibilizados com diferentes tamanhos de entrada e registro. Para realizar a medição, utilizaremos da biblioteca chronos e suas funções.

Avaliamos o desempenho do código nas seguintes situações de arquivo de entrada:

Mil registros com mil caracteres como carga (1000/1000);
Mil registros com cinco mil caracteres como carga (1000/5000);
Cinco mil registros com mil caracteres como carga (5000/1000);
Cinco mil registros com cinco mil caracteres como carga (5000/5000);

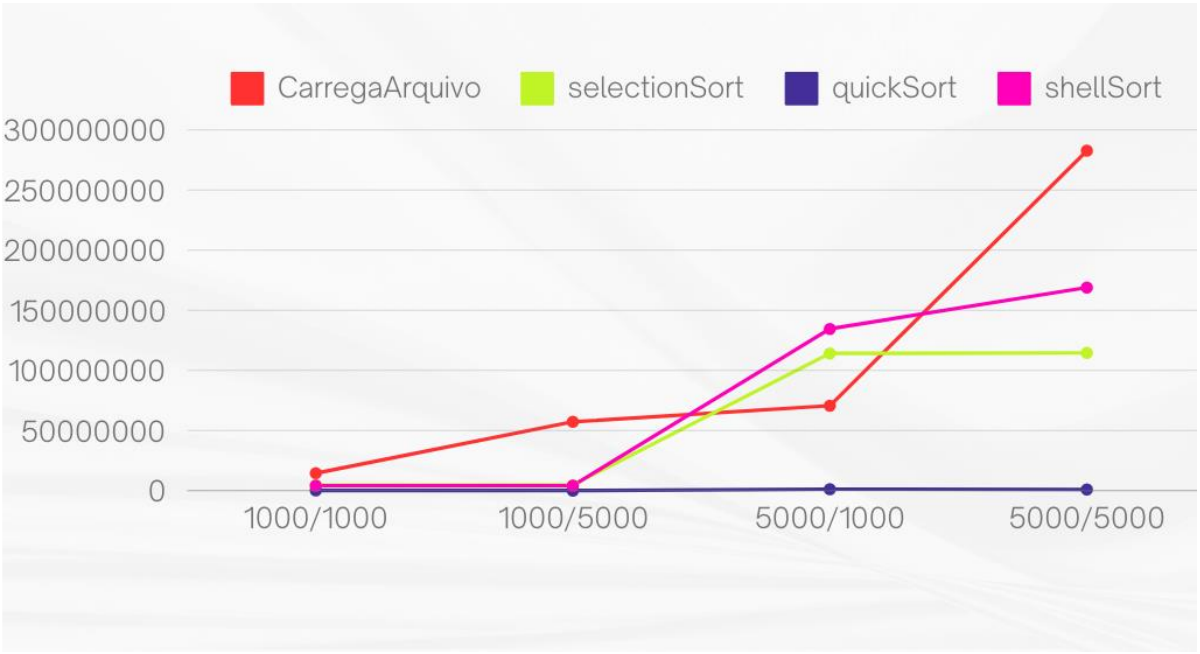

FIGURA 1: GRÁFICO CONTENDO O TEMPO DE EXECUÇÃO DE CADA FUNÇÃO EM NANOSSEGUNDOS, PARA CADA TAMANHO DE ARQUIVO DIFERENTE

Observando o gráfico podemos perceber como as diferentes funções se comportam dependendo do número de pessoas registradas e o tamanho da carga por pessoa. Os algoritmos de ordenação Shell Sort e Selection Sort apresentam desempenho similar, característica esperada devida a ambos serem $O(n^2)$; além disso, ambos se mantém constantes independentemente do tamanho da carga, alterando de maneira significativa apenas quando há um aumento no número de pessoas.

O desempenho do Quick Sort demonstra o porquê desse ser um algoritmo ótimo: como estamos trabalhando com uma grande quantidade de dados desde o início, seu desempenho é incomparável com os algoritmos de tempo $O(n^2)$, tendo seu tempo mais alto na situação de 5000/5000 ainda abaixo de dois milhões de nanossegundos. O desempenho pouco alterado dos algoritmos quando há aumento na carga se deve à ordenação indireta, que nos permite imprimir o arquivo ordenado sem que tenhamos que movimentar a informação diretamente.

O desempenho da função responsável por ler o arquivo se mantém abaixo dos algoritmos $O(n^2)$ na maioria dos casos, se tornando a função mais custosa apenas na situação de 5000/5000. Isso se deve ao fato de CarregaArquivo ser uma função de complexidade $O(n \cdot m)$, a levando a ter um aumento considerável conforme n e m crescem em conjunto.

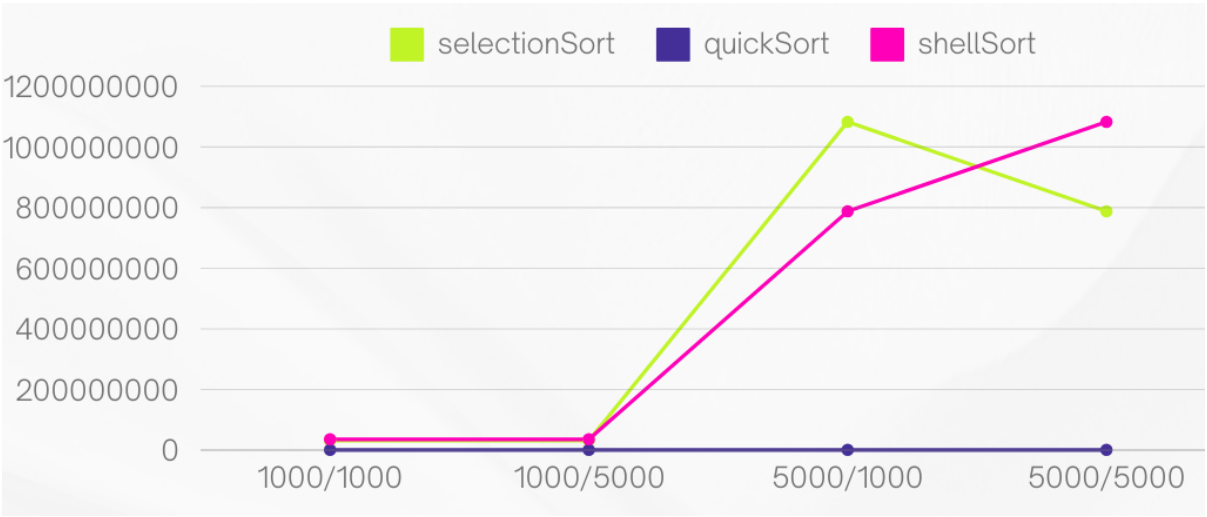

FIGURA 2: GRÁFICO CONTENDO A QUANTIDADE DE IRS PARA CADA FUNÇÃO, PARA CADA TAMANHO DE ARQUIVO DIFERENTE

Ao analisar o programa utilizando da ferramenta callGrind, podemos entender mais da localidade de referência do programa. O gráfico acima demonstra a quantidade de IRs por função, e fica perceptível que um padrão similar ao de desempenho de tempo se repete: shellSort e selectionSort, sendo algoritmos de ordenação mais custosos, se mantém acima do quickSort e sobem para valores imensos quando a quantidade de pessoas aumenta. Já o quickSort se mantém consistentemente baixo, começando abaixo de um milhão e chegando a ser ignorado pelo callGrind para arquivos maiores, tamanha a dominância dos outros algoritmos. Como é de se esperar, o padrão se mantém para outras métricas relacionadas à localidade de referência.

6. Conclusão

Este trabalho lidou com o problema de ordenar arquivos xcsv com grande número de cadastros e informações utilizando diferentes chaves, e a abordagem utilizada para o solucionar foi o de ordenação indireta. Utilizando dessa abordagem, pode-se verificar que o desempenho do programa é consistentemente melhor, e a quantidade de dados atrelada a cada pessoa influencia menos no desempenho do algoritmo.

A resolução desse trabalho permitiu praticar os conceitos previamente ensinados para ordenação de informações, análise de complexidade, avaliação de desempenho e localidade de referência. Alguns aspectos desafiadores da implementação foram implementar os algoritmos de ordenação utilizando a ordenação indireta, além de fazer a análise de complexidade e ver como ela se comparava com a avaliação experimental. Os resultados obtidos alinharam com a análise de complexidade e foram satisfatórios visto o problema que buscávamos solucionar e a abordagem escolhida.

7. Bibliografia

Chaimowicz, L. and Prates, R. Slides virtuais da disciplina de estruturas de dados. Disponibilizados via Moodle. Departamento de Ciência da Computacão. Universidade Federal de Minas Gerais. Belo Horizonte.

Lacerda, A. and Meira, W. Slides virtuais da disciplina de estruturas de dados. Disponibilizados via Moodle. Departamento de Ciência da Computacão. Universidade Federal de Minas Gerais. Belo Horizonte.