Trabalho Prático da disciplina de Estrutura de Dados

Leticia Ribeiro Miranda
2021095686
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil
leticia-ribeiro98@hotmail.com

1. Introdução

O trabalho tem como objetivo a implementação e análise de três algoritmos de ordenação, QuickSort, InsertionSort e BubbleSort. Todos os métodos foram implementados na classe OrdInd, no qual foi projetada para carregar os dados de um arquivo e ordená-los de acordo com o algoritmo e atributo escolhido. Por fim, foi feita a análise experimental, que inclui medições de tempo de relógio e recursos, além de uma avaliação de localidade e referência.

2. Método

O programa foi desenvolvido na linguagem C++, compilado pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificações:

- Sistema operacional: Ubuntu 24.04.1 LTS
- Compilador: gcc (Ubuntu 13.2.0-23 ubuntu 4) 13.2.0
- Processador: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
- RAM: 16Gb (utilizável: 13,9 GB)

2.1. Estrutura de Dados

Para resolver o problema proposto, a implementação apresenta uma classe OrdInd, que organiza e manipula dados estruturados, permitindo que eles sejam carregados de um arquivo e ordená-los com base em três algoritmos de ordenação - QuickSort, InsertionSort e BubbleSort. A escolha desses métodos visa comparar o desempenho de uma abordagem mais simples (InsertionSort e BubbleSort), com uma mais eficiente (QuickSort), possibilitando a análise do comportamento de cada um em diferentes cenários.

O código foi organizado em três arquivos: *OrdInd.hpp*, que contém a definição da classe com seus atributos e métodos; *OrdInd.cpp*, que implementa os métodos definidos no arquivo de cabeçalho; main.cpp, que abriga a lógica para a utilização da classe *OrdInd*.

Na primeira parte do arquivo *OrdInd.hpp*, estão definidos os atributos privados da classe, enquanto na segunda parte são apresentados os construtores, destrutores e métodos públicos. O programa utiliza um array para armazenar os nomes das colunas dos registros e uma matriz para armazenar os valores dos registros.

O arquivo *OrdInd.cpp* é responsável por implementar a classe, seu construtor inicializa os contadores de atributos e dados. O método Swap troca os dados das colunas entre daus linhas. Já o método CarregaArquivo tem a função de ler o arquivo xcsv, preenchendo os

arrays de atributos e dados, verificando limites e formatando corretamente as colunas e linhas. O algoritmo de ordenação QuickSort é implementado por meio de três funções: *Particao*, que realiza a divisão dos dados; *Ordena*, que executa a ordenação recursiva sobre as partições; e *QuickSort*, que serve como interface para ordenar os registros com base em um atributo específico. Já os demais algoritmos de ordenação são implementados de forma independente, cada um em uma única função, InsertionSort e BubbleSort.

Além das operações de ordenação, a classe inclui o método *ImprimeOrdenadoIndice* para imprimir os dados ordenados com base no atributo escolhido. Por fim, os métodos de acesso (getters) incluem funções para obter informações específicas sobre os dados.

Por último, no arquivo main.cpp, o código utiliza três iterações separadas: o primeiro 'for' chama o QuickSort, o segundo chama o InsertionSort e o terceiro chama o BubbleSort. Em cada um desses laços, o método de ordenação correspondente é chamado, seguido pela função de imprimir os dados ordenados.

3. Análise de Complexidade

Abaixo, é identificando a complexidades das funções:

- **Construtor:** Complexidade de tempo e espaço é O(1), pois apenas inicializa variáveis com valores fixos e aloca espaço para as variáveis primitivas.
- **Destrutor:** Complexidade de tempo e espaço é O(1), pois não há operações pesadas no destruidor e também não existe alocação dinâmica de memória.
- **Swap:** Cada troca de linha exige uma troca dos valores dos atributos entre as linhas i e j. Como o número de atributos é n, então ele é proporcional a ele. Logo, a complexidade de tempo é O(n), onde n é o número de atributos. Já a complexidade de espaço é O(1), visto que apenas uma variável temp é usada para armazenar temporariamente um valor durante a troca.
- **CarregaArquivo:** O método faz uma leitura do arquivo e armazena os dados em uma estrutura de dados de m linhas e n atributos, o que leva O(m * n) operações. Além disso, os dados armazenados em uma estrutura bidimensional, logo sua complexidade de tempo e espaço é O(m * n).
- **QuickSort:** Para essa ordenação existem três possibilidades de complexidade:
    - Pior caso: Ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo. Isto faz com que o método *Ordena* seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada. Logo, a complexidade de tempo é $O(n^2)$. Já a complexidade de espaço é O(n), visto que o Quicksort acaba fazendo chamadas recursivas, o que resulta em uma profundidade de pilha igual a n.
    - Melhor caso: O melhor caso ocorre quando cada partição divide o conjunto em duas partes iguais. Portanto, a complexidade de tempo é O(n log n). Já a complexidade de espaço é O(log n), pois a cada divisão, a recursão é chamada para duas sublistas, mas a pilha de chamadas não cresce além de log n níveis.
    - Caso médio: de acordo com Sedgewick e Flajolet (1996, p. 17), a complexidade de tempo é aproximadamente 1,386n log n – 0,846n. Já a de espaço continua O(log n).

- **InsertionSort:** Independente do cenário, ao realizar a ordenação, o Insertionsort armazena temporariamente cada linha de dados, o que requer O(n) de espaço para cada dado, resultando em complexidade espaço igual a O(n). Já na complexidade de tempo, existem dois cenários:
    - Pior caso e caso médio: O algoritmo percorre todos os dados e, para cada dado, move todos os dados anteriores para a direita, resultando em uma complexidade de tempo quadrática, $O(n^2)$.
    - Melhor caso: Ocorre quando o array já está ordenado, assim nunca entrará no loop interno. Portanto, a complexidade de tempo é O(n).
- **BubbleSort:** O algoritmo de ordenação tem um loop duplo que percorre todos os dados, comparando e trocando elementos adjacentes, resultando em uma complexidade de tempo quadrática, $O(n^2)$ no pior caso e no caso médio, onde n é o número de elementos na lista. No entanto, no melhor caso, quando a lista já está ordenada, a complexidade é O(n). Já a complexidade de espaço é O(1), pois não há necessidade de alocar espaço adicional, apenas variáveis temporárias.
- **ImprimeOrdenadoIndice:** A complexidade de tempo é O(m * n), pois o método percorre todas as linhas e colunas para imprimir os dados. Já a complexidade de espaço é O(1), visto que não aloca memória adicional para a impressão, apenas usa variáveis auxiliares temporárias.

As operações na função main consistem na chamada das funções que foram analisadas. Dessa forma, a complexidade de tempo e espaço dela é determinada a partir da soma dessas avaliações feitas. Caso a ordenação caia no pior caso, então a complexidade de tempo será $O(n^2)$ e no melhor caso, O(n). Já para a complexidade de espaço, o pior e melhor caso será O(n).

4. Estratégias de Robustez

No desenvolvimento do código, foram implementadas estratégias de robustez para garantir que o programa lidasse com possíveis erros e condições inesperadas, principalmente em relação à abertura e carregamento dos arquivos.

- Verificação de abertura de Arquivo: No método CarregaArquivo, o código verifica se o arquivo foi aberto corretamente, caso contrário, exibe uma mensagem de erro e interrompe a execução.
- Leitura de dados: A leitura dos dados do arquivo é realizada com verificações de erro após cada operação, para garantir que o arquivo está sendo lido corretamente. Se houver alguma condição inesperada durante a leitura de atributos ou dados, o programa gera uma mensagem e interrompe a execução.
- Limitação de tamanho: No método CarregaArquivo, é limitado o número de atributos e dados, garantindo que o programa não tente alocar mais dados do que o sistema pode lidar, evitando estouros de memória ou falhas.
- Limitação no algoritmo de ordenação: No caso dos algoritmos de ordenação, embora não haja blocos explícitos de captura de exceções, o código mantém a integridade dos índices durante as operações de troca, prevenindo acessos indevidos ou fora de limite.
- Verificação de argumentos na linha de comando: Antes de continuar com o carregamento de dados, o algoritmo verifica se o número correto de argumentos foi informado na linha de comando.

No geral, essas práticas ajudaram a garantir que o código fosse mais coeso, livre de erros e eficiente.

5. Análise Experimental

Na etapa de análise experimental foram utilizadas três bibliotecas para medir o tempo de relógio e recursos e a localidade e referência do programa.

Para medir o tempo de relógio, foi usada a função high_resolution_clock::now() da biblioteca chrono, que registra o tempo inicial e o tempo final ao redor da execução da função alvo. A diferença entre esses dois instantes é calculada e armazenada em uma variável do tipo duration<double, std::milli>, que converte a duração para milissegundos.

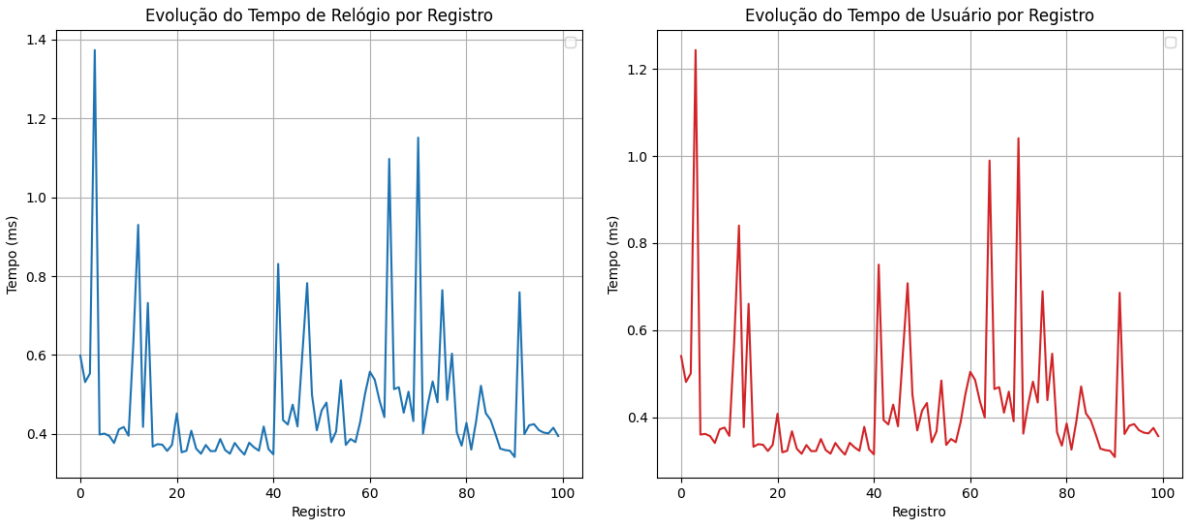
Já para os tempos de utilização de recursos (usuário e sistema), usou-se a função clock_gettime que pertence à biblioteca padrão do sistema no Linux, time.h. Calculou-se a diferença entre o início e fim da execução da função alvo para obter as durações.

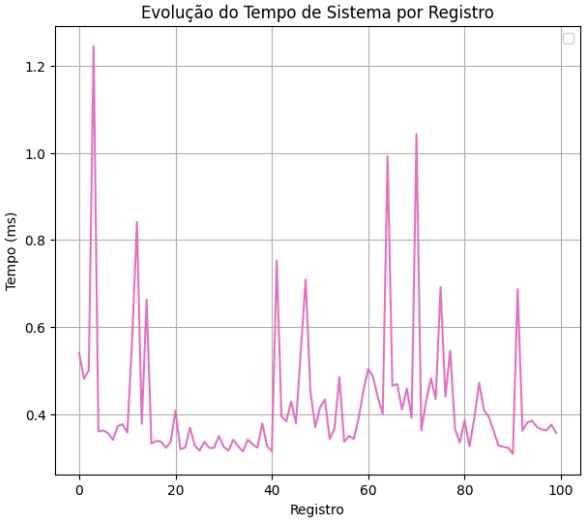
Além disso, foi analisado o tempo de execução dos algoritmos de ordenação utilizando uma base de dados com 100 registros, totalizando 100 mil caracteres. Após esse limite, o programa apresentou um erro de segmentation fault, o que restringiu a quantidade de dados analisados.

Por fim, para avaliar qualitativamente o programa em termos dos acessos de memória e localidade de referência, foi usado a biblioteca memlog.

5.1. QuickSort

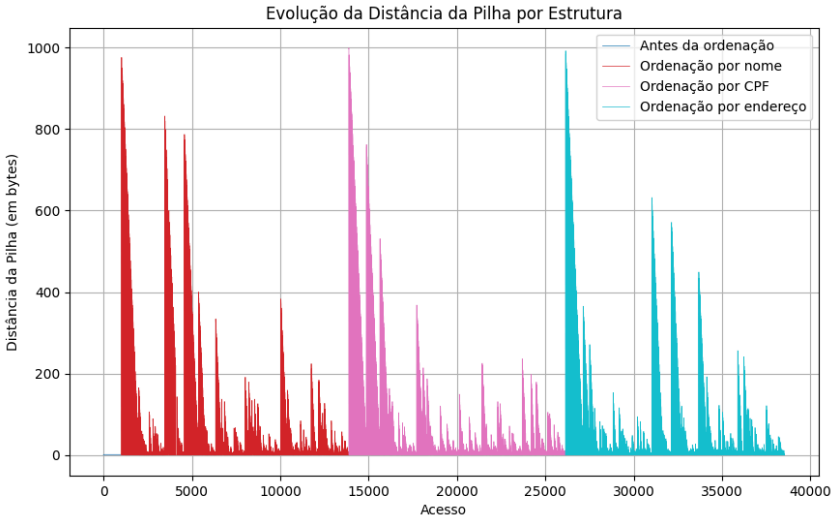
No gráfico abaixo mostra a evolução do tempo de relógio, usuário e sistema do QuickSort. Em entradas pequenas, pode ser difícil ver a complexidade do algoritmo, mas, na teoria, à medida que o arquivo é ordenado, há um crescimento exponencial ou logarítmica nos tempos de execução.



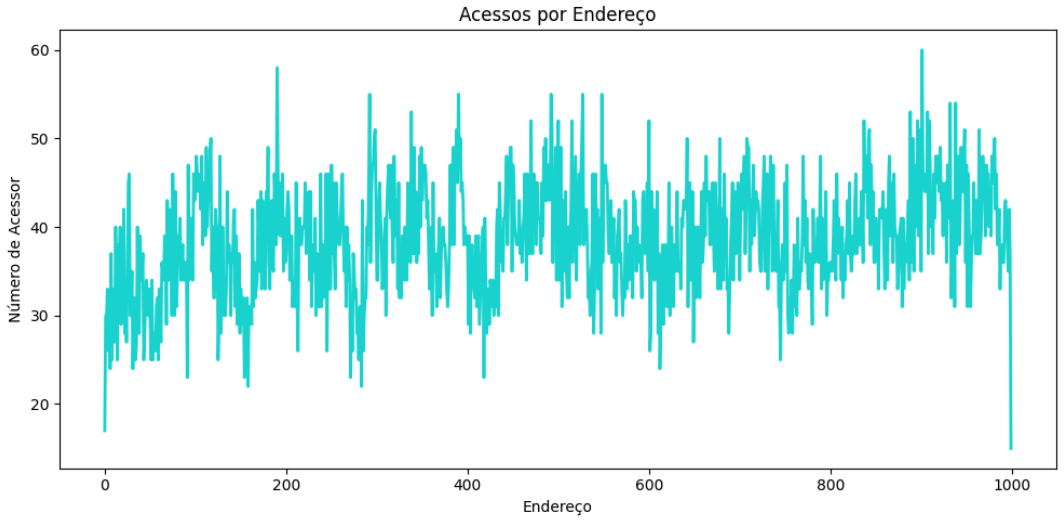


Para analisar a localidade de referência do QuickSort abaixo, foi utilizado uma carga de 1000 registros e 5 milhões de caracteres.

No próximo gráfico, é apresentada a evolução da distância da pilha por tipo de ordenação ao longo dos acessos dos endereços. Antes da ordenação, a distância era mínima, pois os dados ainda estavam organizados de forma original, sem reordenamento. Durante a ordenação, observa-se alguns picos na distância entre os elementos da pilha, no entanto esses são poucos em comparação com outros métodos, mostrando que o QuickSort de fato é eficiente.

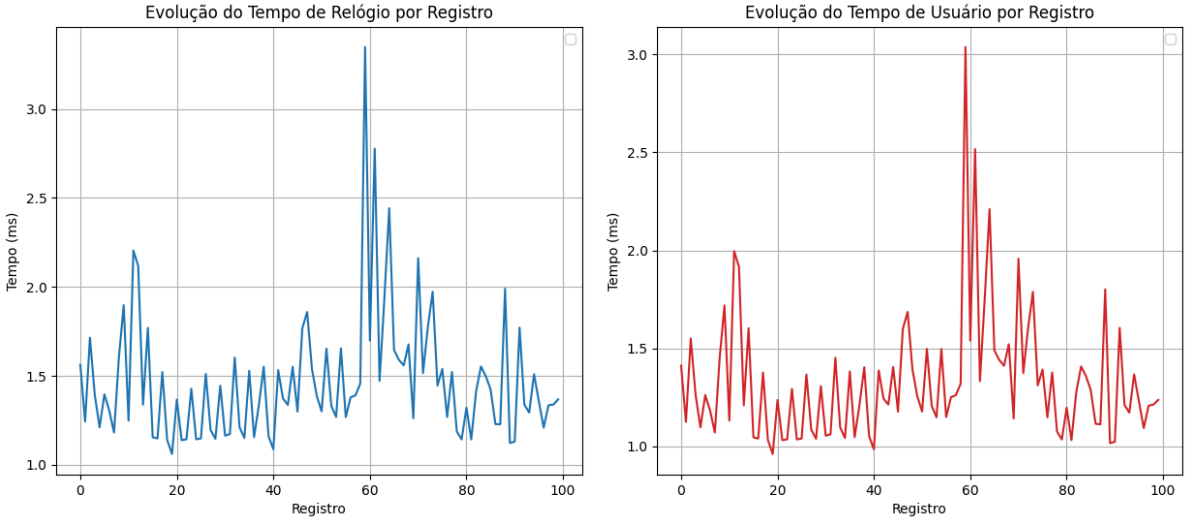


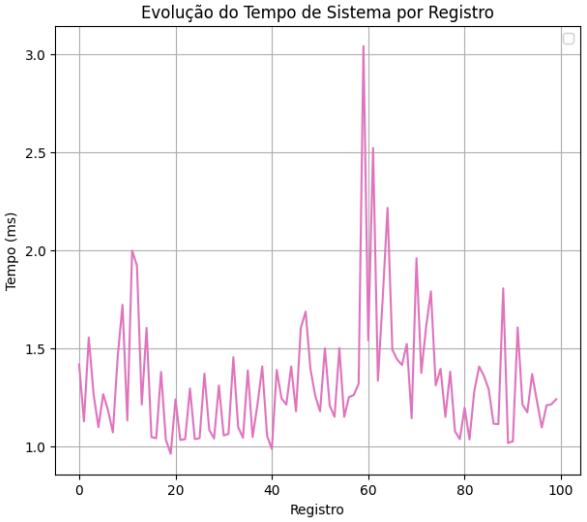
Por fim, a análise a seguir mostra que, em média, cada memória é acessada 38 vezes ao longo da execução do programa. Esse número é relativamente baixo em comparação com outros algoritmos de ordenação implementados, o que era esperado, já que programas com bom desempenho geralmente realizam menos trocas de posições.



5.2. InsertionSort

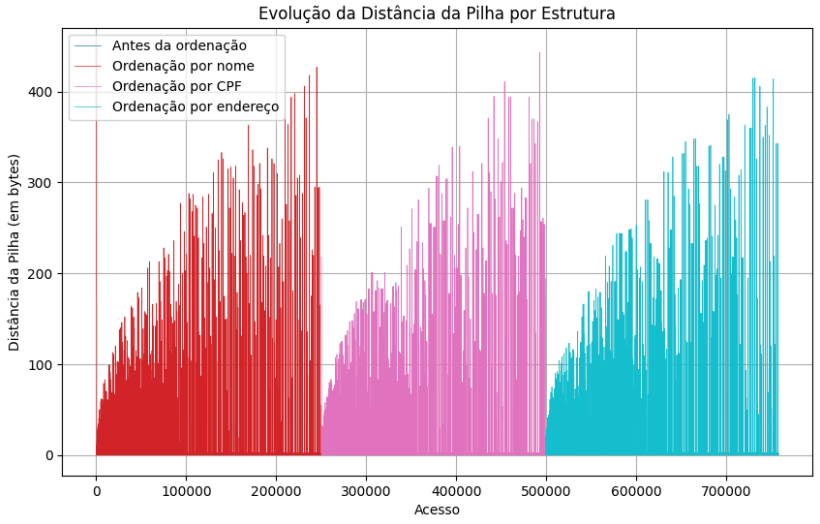
No gráfico abaixo mostra a evolução do tempo de relógio, usuário e sistema do InsertionSort. Em entradas pequenas, pode ser difícil ver a complexidade do algoritmo, mas, na teoria, à medida que o arquivo é ordenado, há um crescimento exponencial ou linear nos tempos de execução.



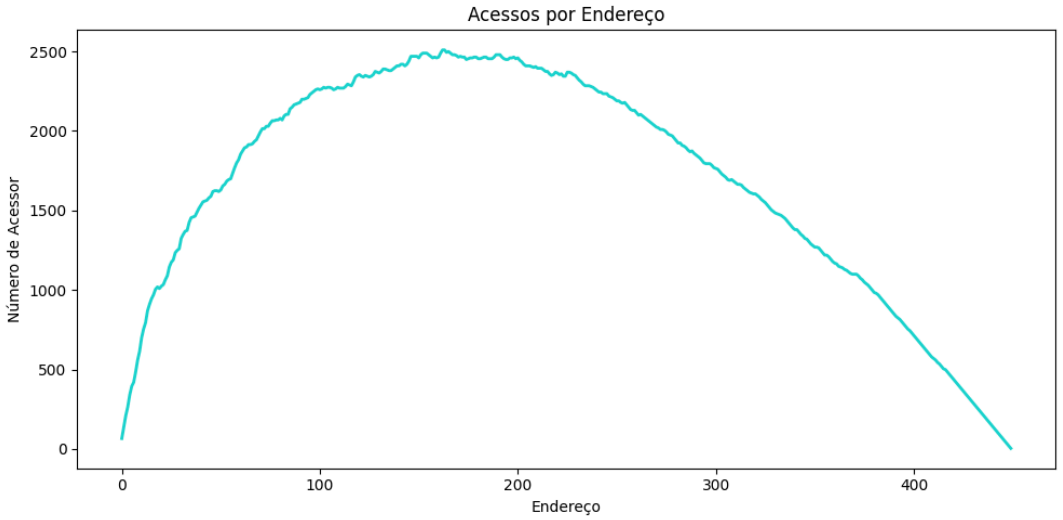


Para analisar a localidade de referência do InsertionSort abaixo, devido à alta quantidade de leituras e gravações de memória, o arquivo de saída do memlog gera um volume significativo de dados. Por esse motivo, foi utilizada uma carga de 450 registros e 500 mil caracteres.

No próximo gráfico, é apresentada a evolução da distância da pilha por tipo de ordenação ao longo dos acessos dos endereços. Antes da ordenação, a distância era mínima, pois os dados ainda estavam organizados de forma original, sem reordenamento. Durante toda a ordenação, a distância da pilha foi alta na maioria dos acessos, o que era esperado, uma vez que o algoritmo realiza diversos deslocamentos de elementos, resultando em uma distância contínua e alta durante o processo de ordenação.

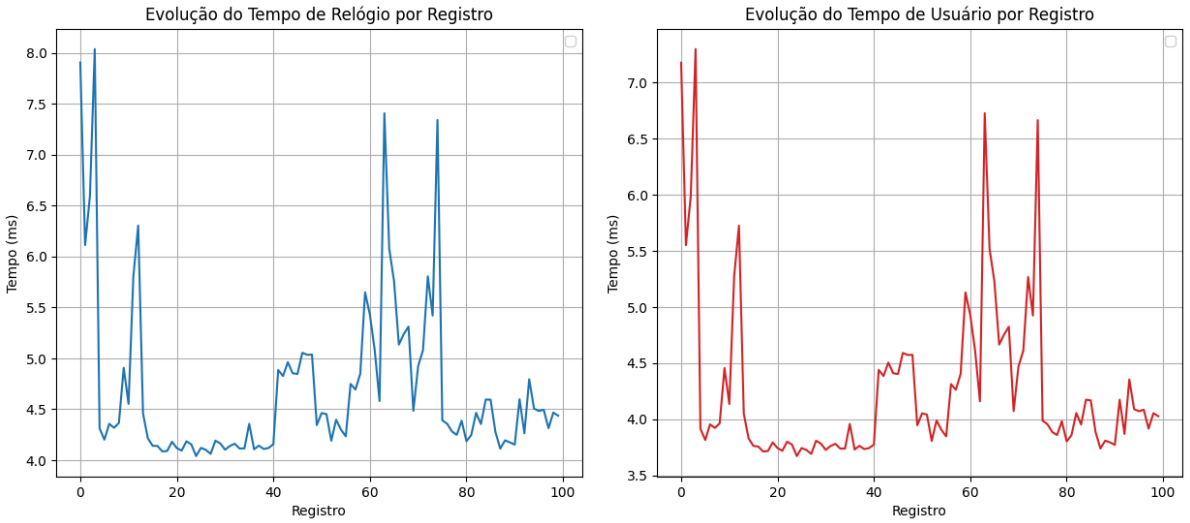


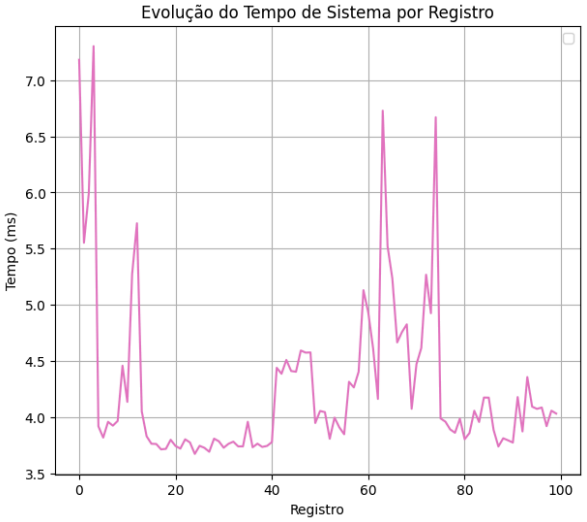
Por fim, a análise a seguir mostra que, em média, cada memória é acessada 1683 vezes ao longo da execução do programa. Esse número é relativamente alto em comparação com outros algoritmos de ordenação implementados, o que era esperado, já que o algoritmo em registros não ordenados sempre entra no loop interno.



5.3. BubbleSort

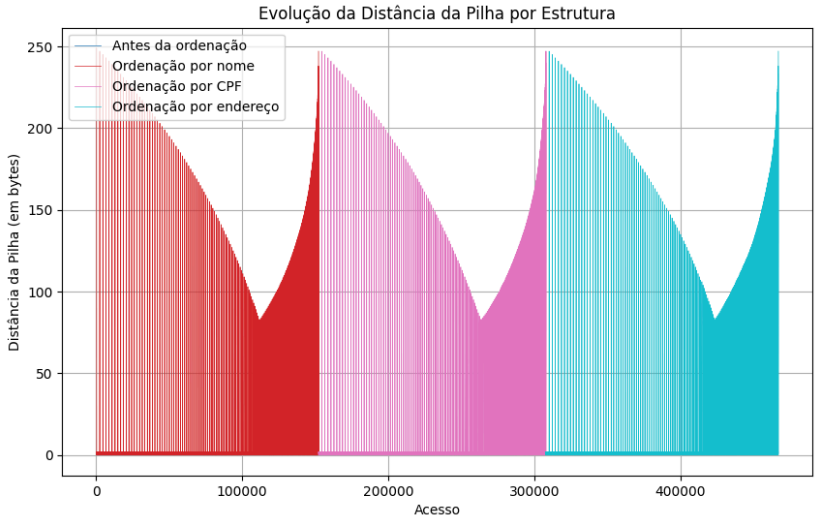
No gráfico abaixo mostra a evolução do tempo de relógio, usuário e sistema do BubbleSort. Em entradas pequenas, pode ser difícil ver a complexidade do algoritmo, mas, na teoria, à medida que o arquivo é ordenado, há um crescimento exponencial ou linear nos tempos de execução.



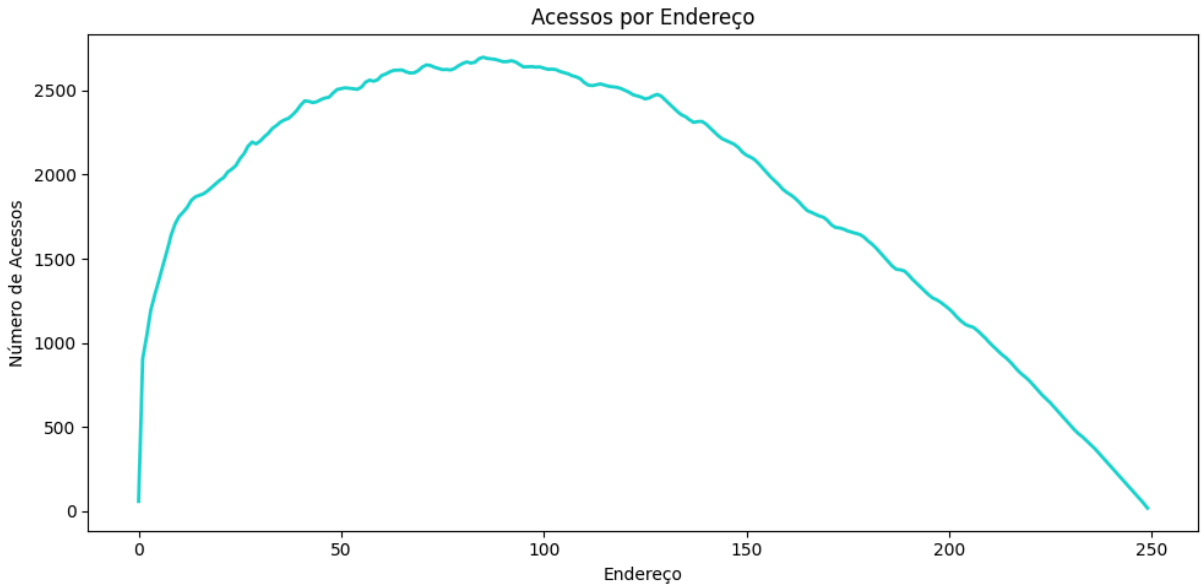


Para analisar a localidade de referência do BubbleSort, foi utilizado uma carga de 250 registros e 300 mil caracteres. Devido à natureza do algoritmo, que realiza muitas trocas entre os elementos, a máquina não conseguiu suportar um número maior de cargas sem comprometer o desempenho.

No próximo gráfico, é apresentada a evolução da distância da pilha por tipo de ordenação ao longo dos acessos dos endereços. Antes da ordenação, a distância era mínima, pois os dados ainda estavam organizados de forma original, sem reordenamento. Enquanto ordenava, a distância da pilha permaneceu alta e constante em quase todo o processo, o que era esperado, já que o algoritmo realiza trocas sequenciais entre pares de elementos, resultando em uma distância contínua e elevada durante a maior parte da execução. Ao final, com os registros quase completamente ordenados, a distância da pilha diminui significativamente. Note que mesmo com menos registros na análise experimental, o BubbleSort gerou uma enorme quantidade de acessos, como é apontado no eixo x do gráfico.



Por fim, a análise a seguir mostra que, em média, cada memória é acessada 1868 vezes ao longo da execução do programa. Esse número é extremamente alto em comparação com outros algoritmos de ordenação implementados, já que, mesmo com um número reduzido de cargas, ele realiza mais trocas que o QuickSort e o InsertionSort.



6. Conclusões

Neste trabalho, foi realizada a implementação e análise dos algoritmos de ordenação. Através da implementação e comparação dos métodos Bubble Sort, Insertion Sort e Quick Sort, foi observado que a escolha do algoritmo de ordenação tem um impacto no tempo de execução e na distância da pilha.

Embora o Bubble Sort e Insertion Sort apresentem simplicidade, eles são menos eficientes, se comparados ao Quick Sort, que se destacou em cenários com grandes volumes de dados.

Já o InsertionSort apresentou um desempenho mais lento, principalmente em contextos com dados não ordenados, mas ainda assim foi eficiente para pequenos conjuntos de dados. Por outro lado, o BubbleSort se revelou o algoritmo mais ineficiente, principalmente em termos de número de acessos e trocas realizadas, sendo mais adequado apenas para conjuntos de dados muito pequenos ou ordenados.

Este estudo reforça a importância de selecionar o algoritmo adequado de acordo com o tamanho e a natureza dos dados a serem ordenados.

7. Bibliografia

Wagner Meira Jr. (2023). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

8. Extra

O objetivo desse tópico é propor uma heurística que melhore a localidade de referência em relação às impressões da base de dados ordenada por diferentes chaves. A estratégia adotada neste tópico foi agrupar registros que são frequentemente acessados, em ordem descrente, do dado mais acessado para o menos.

Para isso, durante as ordenações, foi calculado a frequência de acesso de cada registro, através de um array de inteiros chamado “frequencia”, onde é incrementado em 1 toda vez que o índice foi acessado em cada uma das ordenações.

Após registrar os dados, as frequências foram organizadas em ordem descrente utilizando o QuickSort, em uma função chamada “OrdenarFrequencia”. Em seguida, na função “ReorganizaDados”, os registros foram reorganizados, movendo as linhas dos dados de acordo com a ordem dos índices do array “frequencia”.

Ao realizar essas mudanças, a localidade temporal e espacial é melhorada, visto que dados acessados recentemente têm maior probabilidade de serem acessados novamente em breve e registros armazenados próximos na memória tendem a ser acessados juntos. Além disso, em listas ou arrays que são frequentemente acessadas e/ou alteradas, ordenar os elementos por frequência de acesso pode reduzir o tempo médio para encontrar os dados.

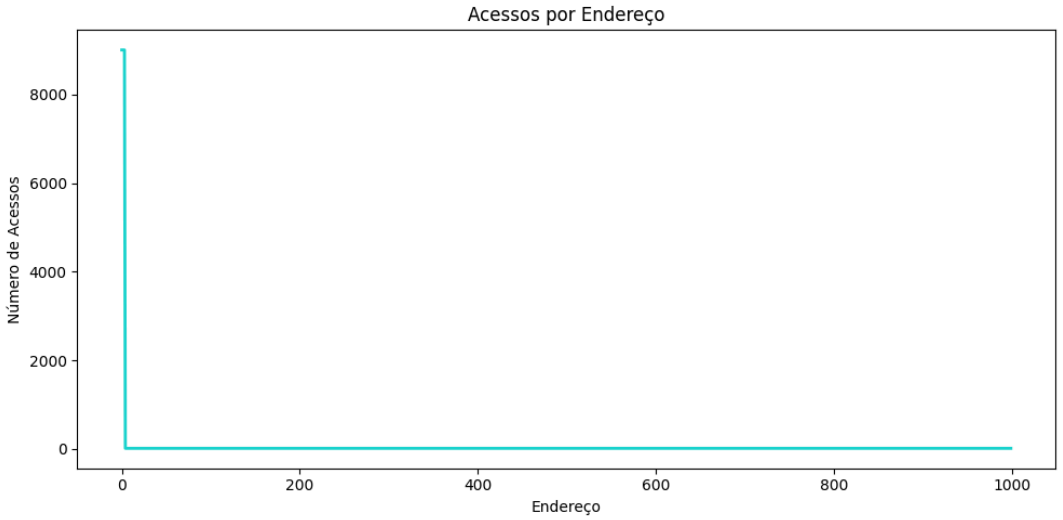
Por outro lado, os sistemas computacionais utilizam cachês para armazenar dados acessados frequentemente. Assim, ordenar os dados por frequência aumenta a probabilidade dos dados mais importantes estarem na memória cache, reduzindo latências causadas por acessos à memória principal.

E por último, estruturas como árvores de busca adaptativas ou listas auto ajustáveis se beneficiam de dados ordenados por frequência, pois isso minimiza o esforço necessário para reorganizar as estruturas durante os acessos.

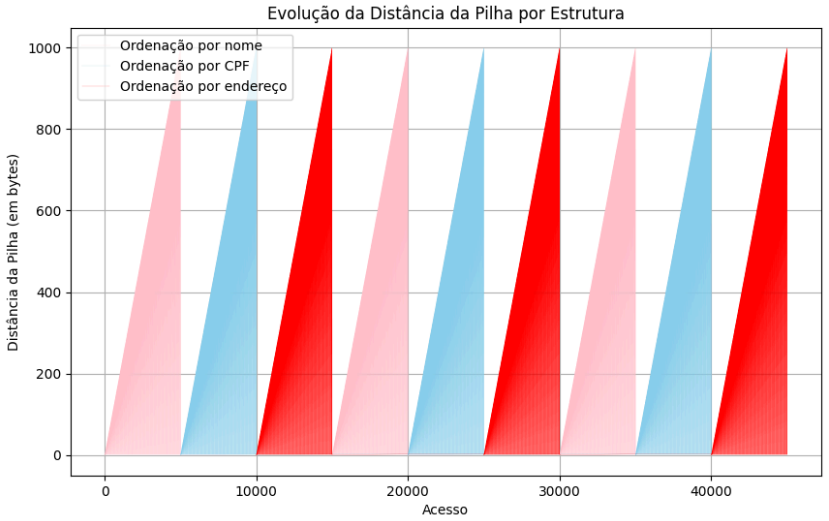
8.1. Análise da Localidade de Referência

Para análise de localidade e referência do QuickSort, foi utilizado uma carga de 1000 registros e 5 milhões de caracteres.

Quanto ao número de acessos, ao ordenar os dados com base na frequência de uso, cada registro é acessado apenas uma vez, o que já era o esperado.



Em relação a distância da pilha, todas as linhas indicam que, à medida que o número de acessos aumenta, a distância da pilha também cresce linearmente para cada tipo de ordenação. Isso sugere que a reordenação não altera o padrão de crescimento da distância da pilha em termos absolutos.



A reorganização dos dados com base na frequência de acesso mostrou-se eficiente para melhorar a localidade de referência. Contudo, o crescimento linear da distância da pilha permaneceu inalterado, indicando que a heurística beneficia acessos frequentes, mas não impacta a estrutura da memória em termos absolutos.

8.2. Bibliografia

Simon, H. A. (1978). A Taxonomy of Heuristic Methods in Problem Solving. Science, 201(4356), 124-128.

MICHALIEWICZ, Z. The Role of Heuristics in Optimization. In: Proceedings of the International Conference on Evolutionary Computation. 1994.