Análise de Desempenho de Algoritmos de Ordenação

Rafael Passos Domingues

I. Introdução

A ordenação numa aplicação computacional é o processo de organizar dados numa determinada ordem. Atualmente, vivemos uma era de grandes volumes de dados e, para isso, a ordenação é essencial, especialmente se tratando de processamento de dados em tempo real [Friedman, 1977].

A ordenação melhora a eficiência da pesquisa de dados específicos dados no computador. Há muitas técnicas sofisticadas para ordenação: Nesta análise, serão investigados os algoritmos de ordenação Bubble Sort, Selection Sort e Insertion Sort aplicados a três distindos arranjos de vetores: em ordem crescente, decrescente e aleatório.

O desempenho desses algoritmos será avaliado com base no número de iterações necessários em cada um dos métodos. No algoritmo foram implementados contadores que acumulam a contagem de cada troca de endereçamento dos elementos, variando o tamanho dos vetores de 100 a 10000, a partir de um incremento de 200 unidades de tamanho.

II. Referencial Teórico

Existem várias métricas que podem ser consideradas para avaliar o desempenho de um algoritmo de ordenação, como o tempo de execução, o consumo de memória e o número de operações realizadas. Também existem vários tipos de algoritmos de ordenação. Cada métrica e algoritmo possui vantagens e desvantagens em relação ao desempenho, dependendo das características dos dados a serem ordenados [Mannila, 1985]; [Chauhan, 2020]; [Harris, 2009].

A seguir, são abordados os métodos a serem explorados neste trabalho.

- **Bubble Sort:** O *Bubble Sort* é um algoritmo que compara repetidamente pares adjacentes de elementos e os troca se estiverem na ordem errada. Esse processo é repetido várias vezes até que a lista esteja totalmente ordenada. O nome *"Bubble"* vem do fato de que os elementos maiores *"borbulham"* para o final da lista durante as comparações e trocas. [Al-Kharabsheh, 2013]
- **Insertion Sort:** O Insertion Sort constrói a lista ordenada um elemento por vez. Ele divide a lista em uma parte ordenada e uma parte não ordenada. Em cada iteração, um elemento da parte não ordenada é selecionado e inserido na posição correta da parte ordenada. [Prajapati, 2017].
- **Selection Sort:** O Selection Sort seleciona repetidamente o menor elemento da lista não ordenada e o coloca no início da lista ordenada. Ele divide a lista em uma parte ordenada e uma parte não ordenada, assim como o Insertion Sort. Em cada iteração, o elemento mínimo da parte não ordenada é identificado e trocado com o primeiro elemento da parte não ordenada [Sedgewick,2011].

III. Material Utilizado

Para realizar esta análise, foi utilizado um dos computadores do laboratório do BCC-UNIFAL com sistema operacional Linux Ubuntu 20.04 LTS. O código foi implementado em C++ e compilado através do ambiente *NetBeans 13*. Por simplicidade, os gráficos foram plotados utilizando as bibliotecas *pandas* e *matplotlib* do Python 3.9 a partir da IDE *VSCode*.

IV. Métodos Implementados

O código *main.cc* em questão implementa os seguintes métodos:

- Inclusão das bibliotecas necessárias: `iostream`, `fstream`, para criar o arquivo de saída `ctime`, para gerar os vetores aleatórios.
- Definição das funções de ordenação: `bubbleSort`, `insertionSort` e `selectionSort`.
- Definição da função `randomArrayGenerator` para gerar um array aleatório a partir de uma sequência ordenada embaralhada aleatoriamente.
- Definição da função `copyArray` para reconstruir os vetores originais e manipulá-los livremente nas chamadas de função, gerando seus distintos arranjos necessários para a análise.
- Definição da função `saveResultsToFile` para salvar os resultados em um arquivo CSV.
- Implementação da função principal (`main`) que realiza as seguintes ações:
  1) Definição de constantes para determinar o tamanho inicial, tamanho final e incremento. *start*, *length* e *step*.
  2) Declaração dos arrays e variáveis necessários para armazenar os resultados: foi criado um vetor *sizes[length]* para armazenar os distintos tamanhos de vetor.
  3) Execução dos algoritmos de ordenação para diferentes tamanhos de arrays (aleatório, crescente e decrescente).
  4) Armazenamento do número de utilizações realizadas em cada algoritmo de ordenação nas variáveis *Usage* para os distintos arranjos de vetores.
  5) Chamada da função `saveResultsToFile` para salvar os resultados em um arquivo *CSV* chamado "sort_usages.csv".

V. Resultados Obtidos

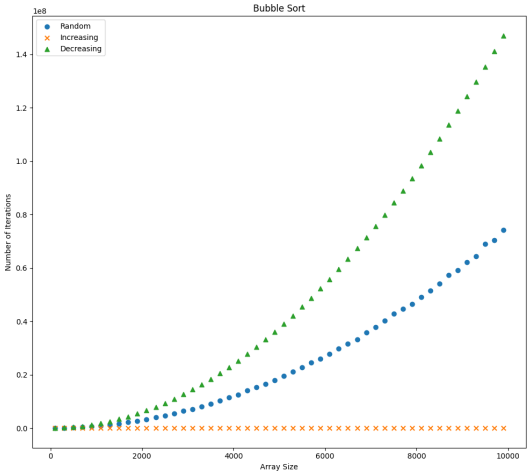

Fig. 1. Número de utilizações do Bubble Sort para diferentes tamanhos de arrays.

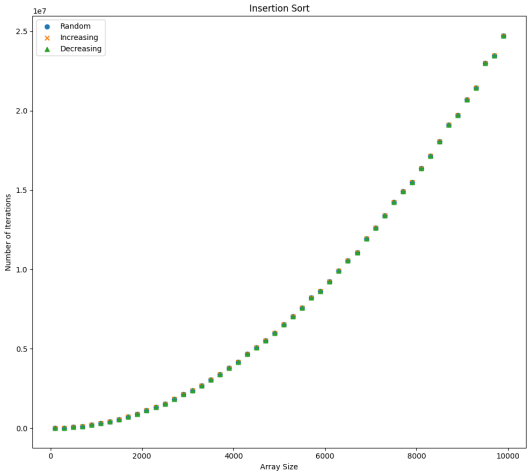

Fig. 2. Número de utilizações do Insertion Sort para diferentes tamanhos de arrays.

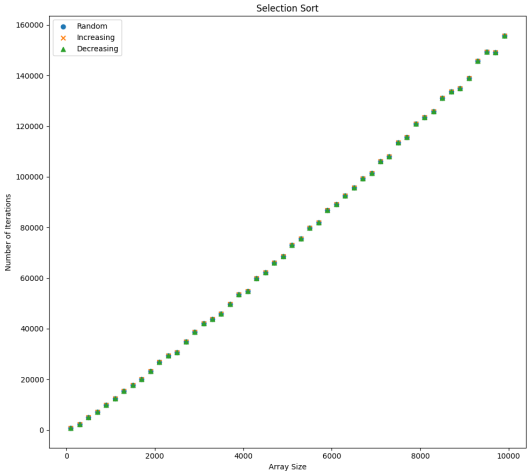

Fig. 3. Número de utilizações do Selection Sort para diferentes tamanhos de arrays.

VI. Conclusão

Foram testadas duas implementações distintas para fazer essa análise de desempenho: a primeira, gerando vetores aleatoriamente sem repetição, elemento por elemento, e a outra, gerando um vetor ordenado e embaralhando aleatoriamente as posições dos elementos.

A implementação que gera um vetor ordenado e depois embaralha aleatoriamente os endereços dos elementos no vetor se mostrou mais performática para esta análise. Houveram dificuldades com a implementação que gera o vetor aleatório elemento por elemento: o tempo de execução foi muito grande, de modo que não foi possível obter resultados, apesar dos testes locais positivos da função que gera o vetor aleatório. Este fato pode ter implicação direta nos resultados deste trabalho.

A escolha de uma determinada técnica de ordenação depende diretamente do arranjo do vetor: Com base nos resultados obtidos, conclui-se que o algoritmo *Bubble Sort* é eficiente para vetores pequenos (< 1000) ou crescentes, e ótimo para vetores decrescentes. Contudo se mostrou bastante ineficiente para vetores grandes e aleatórios. O *Insertion Sort* apresentou um desempenho melhor do que o *Bubble Sort* para vetores crescentes. Já o algoritmo *Selection Sort* não se mostrou tão eficiente quanto o *Insertion Sort*, apesar das implementações serem semelhantes e o fato deste método realizar menos trocas de elementos, não se mostrou tão vantajoso nas situações aplicadas.

VII. Referências Bibliográficas

1) Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software (TOMS), 3(3), 209-226.
2) Mannila, H. (1985). Measures of presortedness and optimal sorting algorithms. IEEE transactions on computers, 100(4), 318-325.
3) Chauhan, Y., & Duggal, A. (2020). Different sorting algorithms comparison based upon the time complexity. International Journal Of Research And Analytical Reviews, (3), 114-121.
4) Satish, N., Harris, M., & Garland, M. (2009, May). Designing efficient sorting algorithms for manycore GPUs. In 2009 IEEE International Symposium on Parallel & Distributed Processing (pp. 1-10). IEEE.
5) Al-Kharabsheh, K. S., AlTurani, I. M., AlTurani, A. M. I., & Zanoon, N. I. (2013). Review on sorting algorithms a comparative study. International Journal of Computer Science and Security (IJCSS), 7(3), 120-126.
6) Prajapati, P., Bhatt, N., & Bhatt, N. (2017). Performance comparison of different sorting algorithms. vol. VI, no. Vi, 39-41.
7) Sedgewick, R., & Wayne, K. (2011). Algorithms (4th Edition). Addison-Wesley Professional.