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. No algoritmo foram implementados contadores a cada troca de endereçamento dos elementos, variando o tamanho dos vetores de 100 a 10000, de 100 em 100.

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.

Existem diversos algoritmos de ordenação disponíveis, cada um com suas características e complexidades. Alguns exemplos comuns incluem o Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort. Cada algoritmo possui vantagens e desvantagens em relação ao desempenho, dependendo das características dos dados a serem ordenados.

A seguir, é abordado 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]. O Bubble Sort possui uma complexidade de tempo de n^2, onde "n" é o número de elementos a serem ordenados. É eficiente para listas pequenas ou quase ordenadas, mas pode ser bastante ineficiente para listas grandes ou desordenadas.
- **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. Esse processo é repetido até que todos os elementos estejam na parte ordenada e a lista esteja totalmente ordenada. O Insertion Sort possui uma complexidade de tempo de n^2 no pior caso, mas pode ter um desempenho melhor do que o Bubble Sort em certas situações, especialmente quando a lista está quase 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. Esse processo é repetido até que todos os elementos estejam na parte ordenada e a lista esteja totalmente ordenada. O Selection Sort também possui uma complexidade de tempo de n^2. Ele geralmente não é tão eficiente quanto o Insertion Sort em termos de número de comparações, mas pode ter menos trocas de elementos, o que pode ser vantajoso em algumas situações [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.

IV. Métodos Implementados

O código em questão implementa os seguintes métodos:

- Inclusão das bibliotecas necessárias: `iostream`, `fstream` e `ctime`.
- Definição das funções de classificação: `bubbleSort`, `insertionSort` e `selectionSort`.
- Definição da função `randomArrayGenerator` para gerar um array aleatório.
- Definição da função `copyArray` para reconstruir os vetores originais.
- 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.
  2) Declaração dos arrays e variáveis necessários para armazenar os resultados.
  3) Preenchimento do array de tamanhos com valores incrementais.

4) Execução dos algoritmos de classificação para diferentes tamanhos de arrays (aleatório, crescente e decrescente).
5) Armazenamento do número de utilizações realizadas em cada algoritmo de classificação.
6) Chamada da função `saveResultsToFile` para salvar os resultados em um arquivo CSV chamado ”sort_usages.csv”.
7) Retorno de 0 para indicar que o programa foi executado com sucesso.

V. Resultados Obtidos

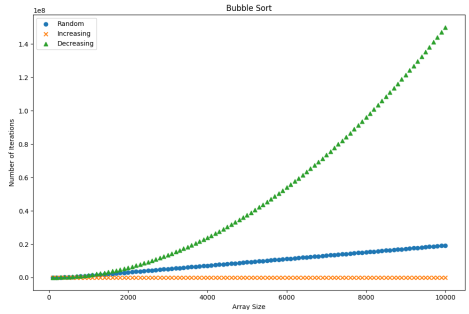

Fig. 1. Número de utilizações do Bubble Sort para diferentes tamanhos de arrays.

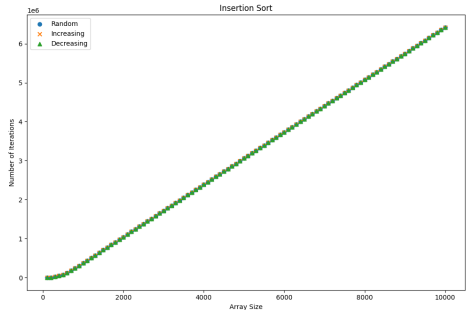

Fig. 2. Número de utilizações do Insertion Sort para diferentes tamanhos de arrays.

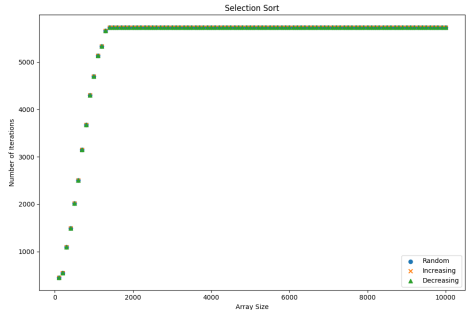

Fig. 3. Número de utilizações do Selection Sort para diferentes tamanhos de arrays.

Os resultados da tabela mostram o número de utilizações de cada algoritmo de ordenação para diferentes tamanhos de arrays. Observa-se que o Bubble Sort tem a maior contagem de utilizações, seguido pelo Insertion Sort e pelo Selection Sort.

VI. Conclusão

A escolha de uma determinada técnica de ordenação depende do tipo de dados: Com base nos resultados obtidos, conclui-se que o método de ordenação por seleção se mostrou o menos performático, pois o número de iterações cresce exponencialmente e estabiliza em um grande número de iterações. O método de inserção é mais linear com o aumento do tamanho do vetor, além de apresentar um comportamento semelhante aos distintos arranjos de vetores. O método *Bubble Sort* se mostrou mais performático com vetores em ordem decrescente, apresentando uma performance razoável com vetores aleatórios e diminuindo a performance em uma relação tamanho ao quadrado do vetor crescente.

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) 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.
3) Prajapati, P., Bhatt, N., & Bhatt, N. (2017). Performance comparison of different sorting algorithms. vol. VI, no. Vi, 39-41.
4) Sedgewick, R., & Wayne, K. (2011). Algorithms (4th Edition). Addison-Wesley Professional.