Análise empírica de métodos de ordenação Insertion Sort e Insertion Sort combinado com Bucket Sort

Davi Marchetti Giacomel[1], **Jaqueline Cavaller Faino**[1], **Maria Eduarda Crema Carlos**[1]

[1]Centro de Ciências Exatas e Tecnológicas – Universidade Estadual do Oeste do Paraná (UNIOESTE) Cascavel – PR – Brazil

{davi.giacomel1, jaqueline.faino, maria.carlos3}@unioeste.br

Resumo. *Nesta pesquisa, exploramos os algoritmos de ordenação Insertion Sort e Bucket Sort, criando um ambiente de testes utilizando C++ e Python. Utilizando este ambiente em um único computador executamos versões seriais do Bucket Sort e do Insertion Sort, analisando o desempenho, medido em tempo de execução, de vários tamanhos de vetor com variadas características de distribuição dos valores. Na implementação do Bucket Sort, utilizamos intervalos iguais para cada balde, de modo que cobrissem os conjuntos de teste do menor ao maior valor. Ao examinar os comportamentos dos algoritmos em conjuntos diversos, observamos desempenhos significativamente melhores em conjuntos ordenados e parcialmente ordenados para o Insertion Sort, e, em geral, que o Bucket Sort tende a performar melhor conforme cresce o número de baldes.*

1. Ambiente de testes e suas métricas

A linguagem selecionada pela equipe para implementar os algoritmos *Insertion Sort* e *Bucket Sort* foi C++. Para a execução dos diversos testes, escreveu-se um *script* na linguagem Python que dispara as execuções dos programas em C++ compilado com diferentes parâmetros, alterando o número de baldes, elementos e o tipo de entrada de dados para o algoritmo. A máquina utilizada possui 16 gigabytes de memóriam RAM e um processador Intel(R) Core(TM) i7-8565U de 1.80GHz. Os conceitos teóricos utilizados e os algoritmos implementados foram obtidos através do material disponibilizado pelo professor da disciplina [Brun 2023] e através do livro *Introduction to Algorithms*[Cormen et al. 2022].

No *Bucket Sort*, uma decisão de projeto importante é a forma como ocorre a distribuição dos elementos do vetor nos baldes. A fórmula utilizada para determinar os intervalos e a partinência de cada elemento a cada balde tem como objetivo homogeneizar a distribuição dos valores do vetor entre os diferentes baldes, considerando também casos em que haja uma homogeneidade na distribuição dos valores do conjunto a ser ordenado. Para determinar o *range* de cada balde, considerando o menor valor do conjunto como $lower_bound$, o maior como $upper_bound$ e o número de baldes como $n_buckets$, aplica-se a seguinte fórmula:

$$range = \lceil \frac{upperbound - lowerbound}{nbuckets} \rceil$$

O número do balde a que um dado elemento m pertence é dado pela expressão

$$\frac{m - lowerbound}{range}$$

2. Análise do comportamento dos métodos durante a execução dos testes

Ao analisarmos o desempenho dos algoritmos de ordenação Insertion Sort puro e Insertion Sort com *Bucket Sort* (com 10, 100 e 1000 baldes) em diferentes tipos de conjuntos de dados, desde aleatórios até parcialmente ordenados, podemos observar certos padrões em seu comportamento e distribuições de dados mais ou menos benéficas para cada. Detalharemos as observações e padrões identificados em cada cenário, proporcionando uma compreensão mais aprofundada do desempenho, em termos de tempo de execução, desses algoritmos.

Foram realizados testes de tipos diferentes de vetores, sendo estes: aleatórios, decrescentes, ordendados e parcialmente ordenados. Além disso, foram utilizados diferentes tamanhos de vetores com cada um desses tipos, sendo eles: 100, 200, 500, 1000, 2000, 5000, 7500, 10000, 15000, 30000, 50000, 75000, 100000, 200000, 500000, 750000, 1000000, 1250000, 1500000 e 2000000.

2.1. Conjunto de dados aleatório

Para o conjunto de dados aleatórios, apresentados na Figura 1 na página 3, o tempo de execução do *Insertion Sort* puro aumenta significativamente à medida que o número de elementos no conjunto cresce. Essa é uma característica esperada do algoritmo, que tem complexidade quadrática, tornando-se menos eficiente para conjuntos de dados maiores. Ao utilizar o *Insertion Sort* com *Bucket Sort* de 10 baldes, há uma melhoria evidente nos tempos de execução em comparação com o *Insertion Sort* puro. O *Bucket Sort* é eficaz para conjuntos aleatórios, pois divide o problema em um número constante de subproblemas menores, se aproximando, conforme aumenta o número de baldes, de um desempenho linear. A complexidade do *Bucket Sort* é linear no melhor caso, o que contribui para tempos de execução mais rápidos.

Aumentar o número de baldes para 100 continua a melhorar o desempenho. Isso ocorre porque mais baldes permitem uma distribuição mais fina dos elementos, reduzindo ainda mais a carga de trabalho do *Insertion Sort* em cada balde. O *Bucket Sort* de 100 baldes supera o de 10 baldes em termos de eficiência. E esse evento ocorre também com 1000 baldes, superando os algoritmos apresentados anteriormente.

2.2. Conjunto de dados decrescentes

O tempo de execução do *Insertion Sort* puro aumenta quadraticamente à medida que o número de elementos no conjunto cresce. Isso ocorre porque, em conjuntos de dados decrescentes, cada elemento precisa ser movido para o início do array, resultando em uma complexidade quadrática. A introdução do *Bucket Sort* com 10 baldes tem uma melhoria considerável em relação ao algoritmo anterior, mesmo com dados distribuidos de forma decrescente, a fórmula que foi utilizada para o *range* de cada balde garante que esses dados são distribuídos de forma uniforme nos baldes.

A melhoria é levemente menos acentuada em comparação com conjuntos de dados aleatórios para o *Insertion Sort* com *Bucket Sort* de 100 baldes, ainda assim o algoritmo auxilia a aliviar o impacto do conjunto, demonstrando um bom resultado no tempo de execução. Usar 1000 baldes melhora ainda mais o desempenho, mas a eficácia diminui em comparação com conjuntos aleatórios. A complexidade do *Bucket Sort* é mais evidente quando os dados estão organizados de forma mais uniforme. Por isso a

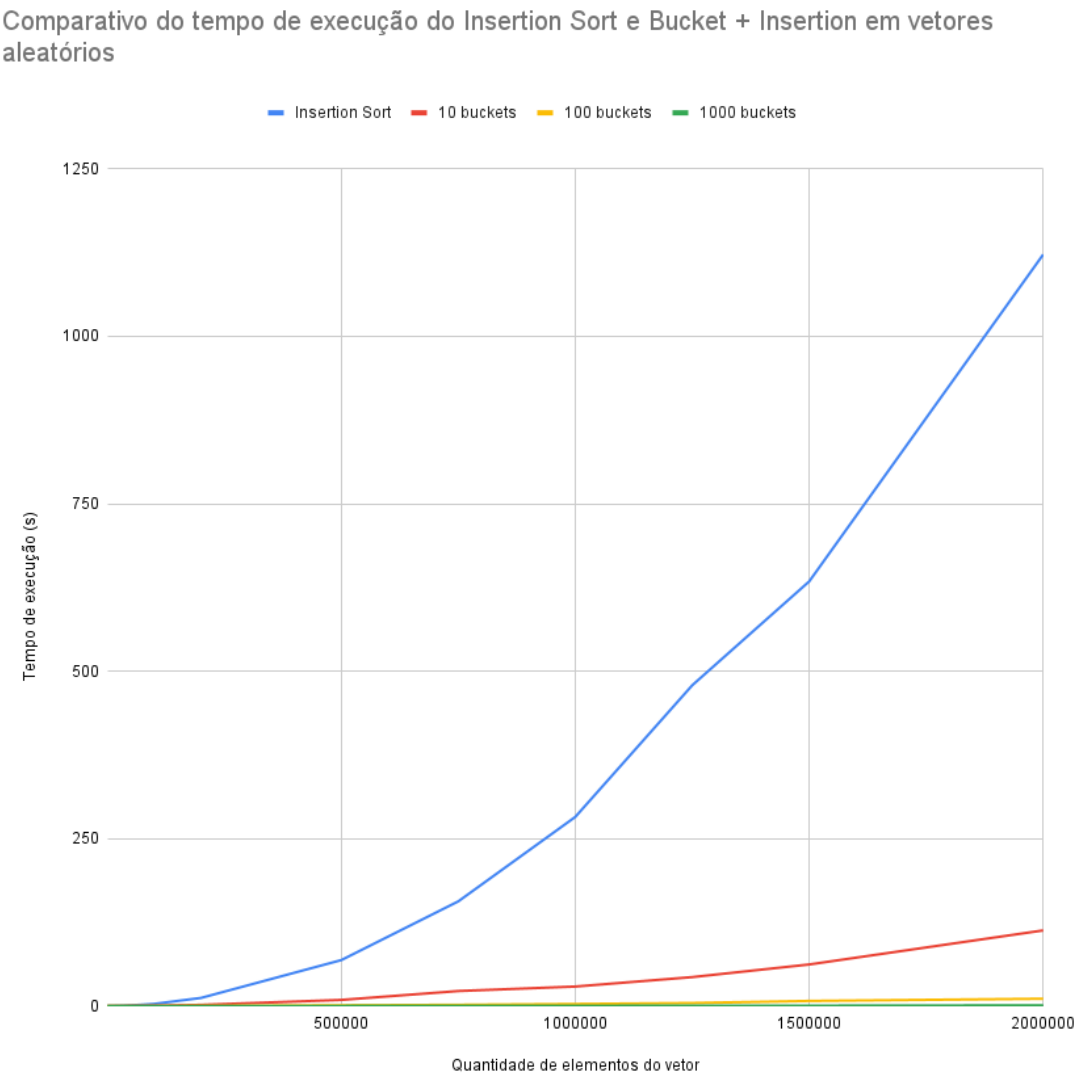

Figura 1. Comparação do tempo de execução do insertion sort e bucket + insertion em conjunto de dados aleatórios

importância de ter um conjunto de dados linear para obter os benefícios do *Bucket Sort*, e a forma utilizada para o *range* auxilia nesse resultado. A figura 2 na página 4 apresenta os resultados discutidos anteriormente.

2.3. Conjunto de dados ordenados

O *Insertion Sort* puro apresenta um desempenho excepcionalmente bom em conjuntos já ordenados, já que neste caso o algoritmo tem desempenho linear, reduzindo a necessidade de trocas e movimentações de dados e o número de execuções do laço interno do algoritmo.

O *Bucket Sort* com 10 baldes não apresenta um benefício significativo quando o conjunto já está ordenado. Na realidade, a sobrecarga introduzida, embora linear, pe-

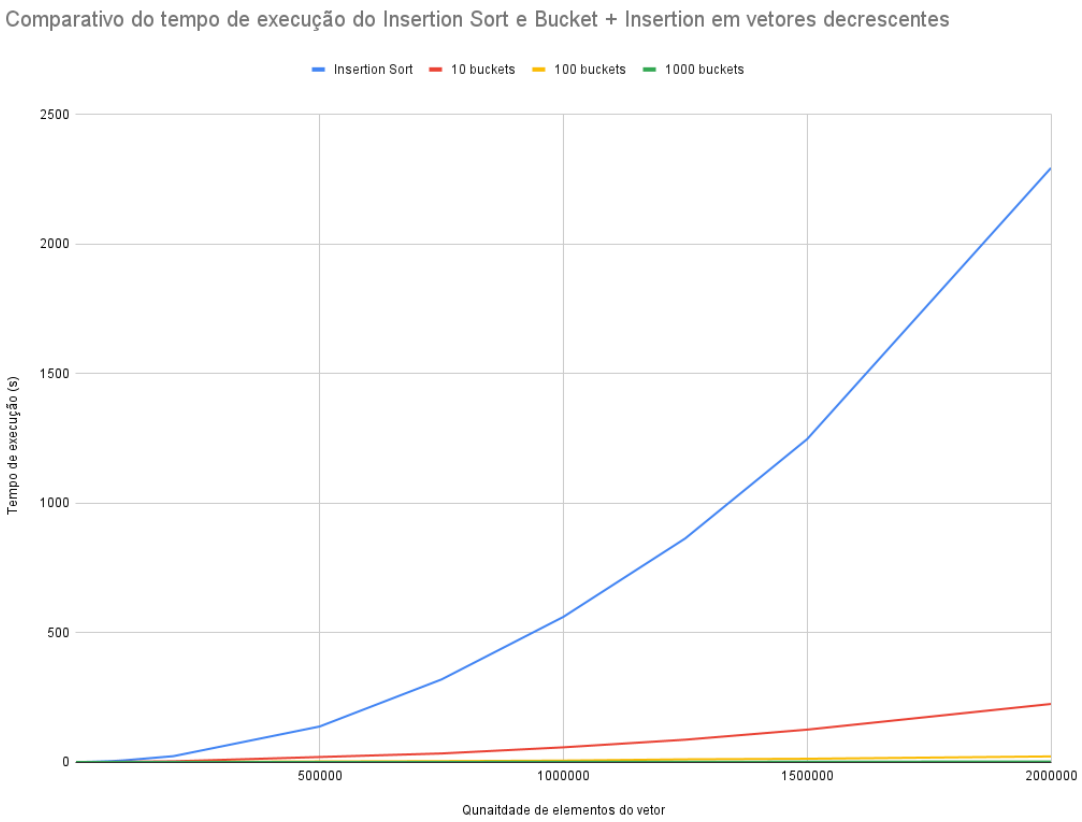

Figura 2. Comparação do tempo de execução do insertion sort e bucket + insertion em conjunto de dados decrescentes

las etapas de *split* e *join* do *Bucket Sort* resulta em comportamentos lineares piores que o comportamento do algoritmo base nesses casos específicos. O desempenho melhora um pouco ao aumentar o número de baldes para 100 e 1000, mas tal melhoria não é tão acentuada em comparação aos conjuntos de dados aleatórios ou decrescentes. O mesmo ocorre com 1000 baldes. A soma dos custos lineares das etapas do *Bucket Sort*, como os dados ordenados, será sempre maior do que a simples execução do *Insertion Sort* para esse tipo de conjunto. Quando o conjunto de dados já está ordenado, o *Insertion Sort* puro é sempre mais eficiente, como podemos observar na figura 3 na página 5, porque ele percorre a lista uma vez e verifica que os elementos já estão na ordem correta. No entanto, ao introduzir o *Bucket Sort*, mesmo que em uma versão otimizada com 10, 100 ou 1000 baldes, há um custo adicional associado à distribuição dos elementos nos baldes e à subsequente concatenação desses baldes. Portanto, ao lidar com conjuntos de dados ordenados, a sobrecarga introduzida pelo *Bucket Sort* pode superar os benefícios que ele traz quando lidamos com conjuntos desordenados, sendo mais eficaz em cenários onde os elementos estão distribuídos de forma mais aleatória.

2.4. Conjunto de dados parcialmente ordenados

O *Insertion Sort* puro tende a ter um desempenho bom em conjuntos parcialmente ordenados, especialmente quando a quantidade de elementos fora de ordem não é muito grande,

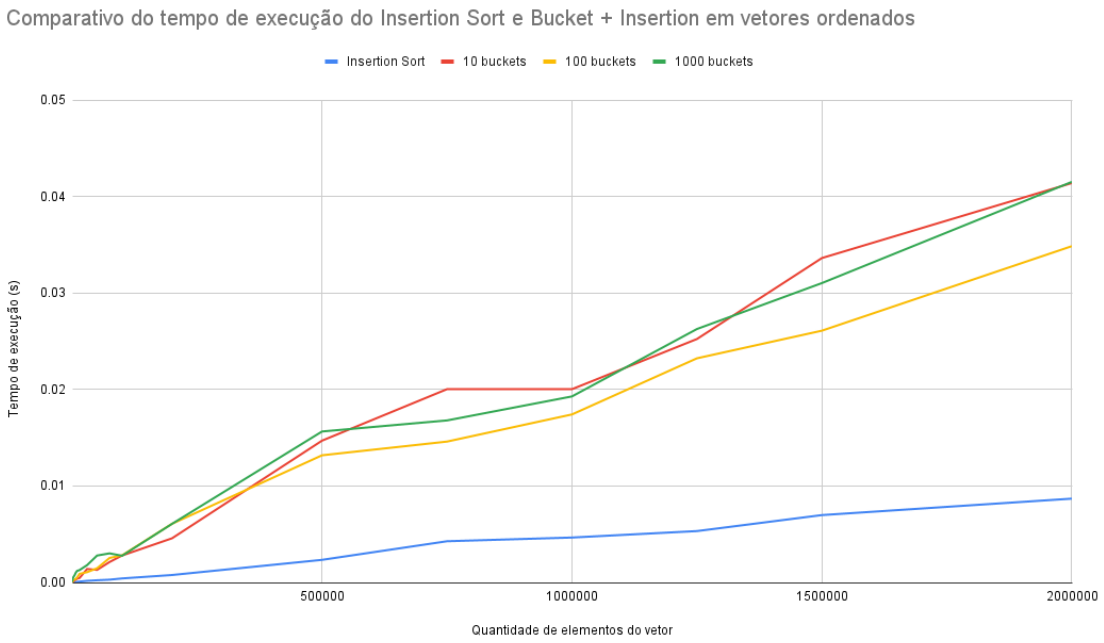

Figura 3. Comparação do tempo de execução do insertion sort e bucket + insertion em conjunto de dados ordenados

tendendo ao tempo de execução linear. Isso ocorre porque o *Insertion Sort* percorre a lista e insere elementos fora de ordem no local correto, o que é eficiente quando há apenas alguns elementos desordenados pois pouca movimentação de dados é necessária.

Observa-se um comportamento semelhante dos algoritmos ao executar sobre os conjuntos de dados ordenados e parcialmente ordenados. Isso se deve ao melhor caso do *Insertion Sort* ocorrer quando o vetor já está ordenado, e vetores parcialmente ordenados incorrem em comportamentos que se aproximam do linear. Entretanto, pela parcialidade da ordenação do conjunto, a divisão em baldes pode gerar subconjuntos mais desordenados do que originalmente estavam antes da fase de *split*, o que pode incorrer em perdas de desempenho em alguns casos, fato que explica parcialmente o comportamento oscilante das curvas. Além disso, por mais controladas as condições de teste, em um intervalo de tempo tão pequeno (0.05 segundos) ruídos provenientes de rotinas do Sistema Operacional são mais significativos.

3. Conclusão

De acordo com os resultados apresentados, é perceptível os casos em que o *Insertion Sort* tem complexidade linear, com algoritmos ordenados ou parcialmente ordenados, como esperado de acordo com o embasamento teórico obtido ao decorrer da disciplina. Além disso, é possível concluir que o uso do *Bucket Sort* é válido para vetores aleatórios e decrescentes de tamanhos superiores a 500000 elementos, onde a complexidade quadrática do *Insertion Sort* se torna mais perceptível.

Em vetores ordenados e parcialmente ordenados, como o *Insertion Sort* se torna linear ou se aproxima desse comportamento, não há muita vantagem em utilizar o *Bucket Sort*, pois o custo de distribuir os elementos em *buckets* é maior que o custo linear puro da

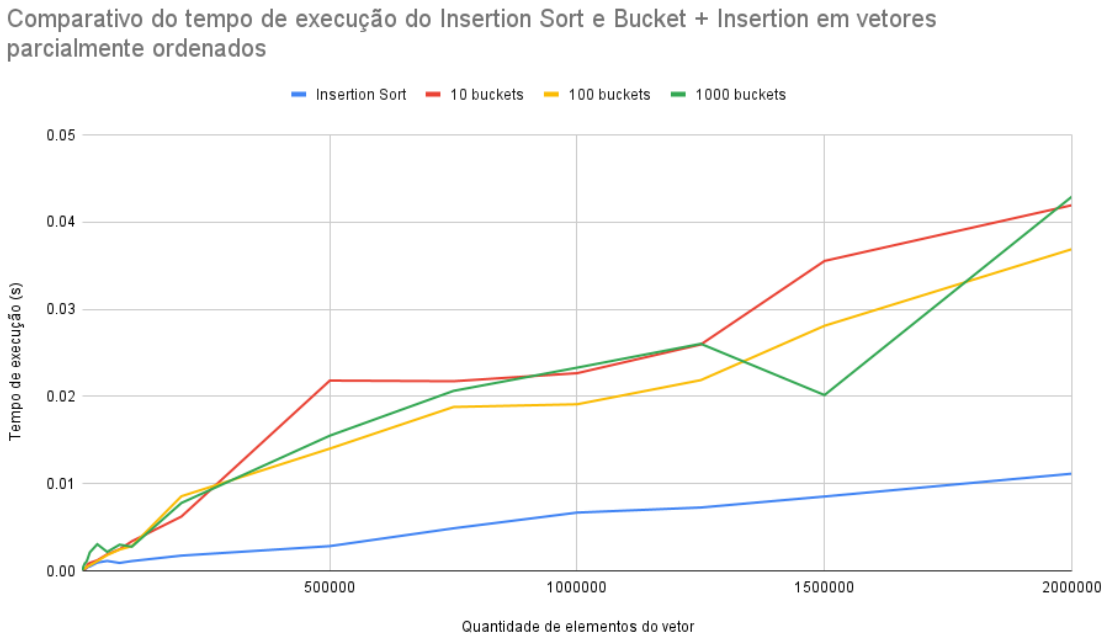

Figura 4. Comparação do tempo de execução do insertion sort e bucket + insertion em conjunto de dados parcialmente ordenados

ordenação do vetor sem passar pela fase de *split*.

Referências

Brun, A. L. (2023). Aula 6 - métodos de ordenação.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to algorithms*. MIT press.