Análise de Complexidade e Experimental de Algoritmos de Ordenação

Bruno Santos de Lima
Faculdade de Ciências e Tecnologia
Universidade Estadual Paulista
Presidente Prudente, Brasil
brunoslima4@gmail.com

Leandro Ungari Cayres
Faculdade de Ciências e Tecnologia
Universidade Estadual Paulista
Presidente Prudente, Brasil
leandroungari@gmail.com

I. Introdução

Diversas aplicações da atualidade envolvem um grande volume de dados, desde aplicações comerciais simples a grande aplicações científicas, todas estão nesse contexto. A organização estrutural de conjunto de dados, além de prover melhor usabilidade, também otimiza tempo e o consumo de recursos, tanto de processamento quanto de memória para a execução.

Neste contexto, este trabalho apresenta uma análise dos principais algoritmos de ordenação, o qual está dividido nas seguintes seções: na Seção 2, são apresentados os algoritmos de ordenação utilizados, indicando a abordagem utilizada no processo juntamente com a sua respectiva análise assintótica. Na Seção 3, são apresentados os estudos comparativos analisando a variações dos conjunto de dados de entrada e abordagens utilizadas na ordenação. Por fim, na Seção 4, são apresentadas as considerações finais do estudo.

II. Algoritmos de Ordenação

A. Algoritmos baseados em Troca

1) Bubble Sort: O presente algoritmo utilizada a abordagem de troca, através da permutação de elementos vizinhos, seguindo a ideia de densidade dos elementos, em que pode-se optar por varrer o arranjo levando o maior elemento (mais pesado) ao fim do vetor, ou conduzir o menor elemento (mais leve) ao início desse. Esse passo (uma das duas opções excludentemente) é realizado sucessivamente para os subvetores não ordenados remanescentes até que o vetor esteja ordenado como um todo.

A abordagem original prevê que todos os elementos adjacentes sejam comparados através de $n-1$ iterações, porém é possível que o arranjo esteja ordenado sem que sejam necessárias todos esses ciclos. De modo a prevenir operações desnecessárias, algumas abordagens utilizam-se de *flags* para a detecção de trocas, caso a última iteração tenha ao menos uma ocorrência, há necessidade de pelo menos mais uma iteração, em caso negativo, o processo pode ser encerrado.

A Figura II-A1 apresenta a abordagem de ordenação utilizada pelo algoritmo referido.

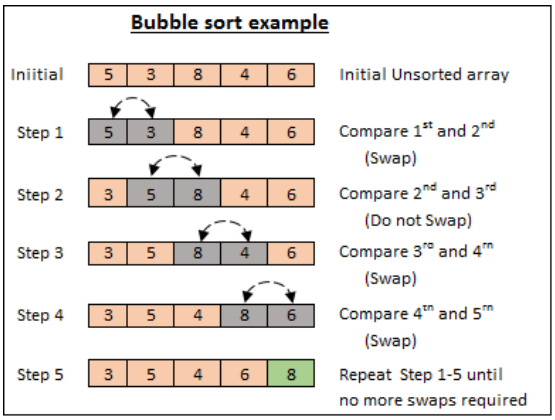

Fig. 1. Modelo de comparação de elementos adjacentes.

Complexidade:

- **Melhor caso:** $O(n)$, para vetor crescente somente na implementação com melhoria.
- **Caso médio:** $O(n^2)$.
- **Pior caso:** $O(n^2)$, para vetor decrescente e aleatório.

2) Quick Sort: O algoritmo foi proposto por C.A.R. Hoare em 1962, tem como estratégia a divisão do arranjo original em partições a partir da determinação de um pivô. Para qualquer abordagem de escolha do pivô, em cada partição busca-se encontrar os elementos maiores que o pivô a partir do início e os menores a partir do fim do arranjo, os quais são trocados de posição aos pares, de modo a ordenar a partição, se necessário também através da quebra de subpartições.

Em relação a abordagem de escolha de pivô, há diversas técnicas que buscam explorar possíveis características dos elementos do arranjo. Dentre as principais, pode-se destacar a escolha do elemento inicial ou final do vetor, o uso de propriedades estatísticas como a média e mediana, entre outras. O principal objetivo de qualquer uma dessas estratégias é minimizar a ocorrência dos piores casos de recorrência, resultando em um comportamento assintótico quadrático.

Geralmente, os piores caso desse algoritmo consistem na escolha de pivô como elemento mínimo ou máximo, para entradas de dados crescentes e decrescentes.

Complexidade:

- **Melhor caso:** $O(nlogn)$.
- **Caso médio:** $O(nlogn)$.
- **Pior caso:** $O(n^2)$, em geral para o pivô mínimo e máximo.

B. Algoritmos baseados em Inserção

1) Insertion Sort: A estratégia de ordenação por inserção consiste em um dos métodos mais simples, sendo extremamente eficiente em conjuntos pequenos, com estratégia de percorrer o esquerdo da esquerda para a direita deixando os elementos ordenados à esquerda. Uma situação cotidiana que aplica a abordagem referida é a inserção de cartas na mão de um jogador, seguindo a estrutura de dados deque.

Complexidade:

- **Melhor caso:** $O(n)$, para arranjo crescente.
- **Caso médio:** $O(n^2)$.
- **Pior caso:** $O(n^2)$.

2) Shell Sort: O algoritmo Shell Sort foi proposto por Donald Shell em 1959, consistindo em um dos métodos de ordenação mais eficientes dentre os modelos quadráticos, baseando-se no Insertion Sort, a partir de comparações com elementos não adjacentes, desse modo, facilitando o deslocamento dos menores elementos para o início do vetor através do uso de saltos regressivos de tamanho.

O grande diferencial desse método é a utilização dos saltos, porém não existe uma abordagem única para o cálculo do tamanho dos saltos, consistindo em um fator determinante na mensuração da complexidade assintótica. Dentre os principais modelos têm-se o n primeiros múltiplos de um fator ou combinação de fatores (ex. 2^p3^q) ou os n primeiros números primos.

A Figura II-B2 apresenta a ordenação das partições de elementos distantes, com saltos de tamanho 40, 13 e 4 respectivamente, partindo do arranjo inicial e alcançando o conjunto ordenado.

Complexidade:

- **Caso médio:** $O(n^{3/2})$

C. Algoritmos baseados em Seleção

1) Selection Sort: O algoritmo Selection Sort, trata-se modelo mais básico de algoritmo de ordenação, utilizada muitas em ambientes naturais e é mais intuitivo para os seres humanos, pois se utiliza da abordagem de força bruta, sem aproveitar nenhuma peculiaridade do conjunto de dados ou

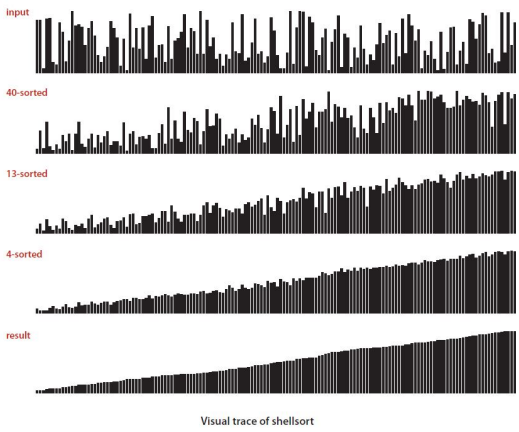

Fig. 2. Modelo de partições com elementos distantes em diferentes tamanhos.

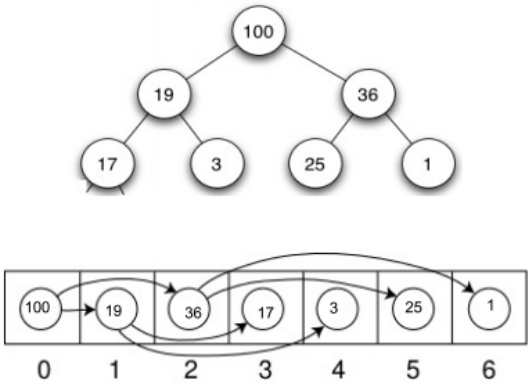

Fig. 3. Equivalência entre a árvore e arranjo.

vantagem que possa ser tomada. Sua estratégia consiste em percorrer todo o arranjo n, comparando todos os elementos de forma a encontrar o menor, em seguida, o segundo menor e assim por diante, de modo a organizar todo o conjunto.

O grande destaque para esse algoritmo, dentro do contexto das abordagens quadráticas, consiste no reduzido número de trocas, que no máximo ocorre n vezes.

Complexidade:

- **Caso médio:** $O(n^2)$, de modo invariante.

2) Heap Sort: O algoritmo Heap Sort, proposto por J.W.J. Williams em 1964, possui uma das implementações mais sofisticadas baseando-se em uma fila de prioridades, através da utilização de um vetor para a representação de uma árvore binária, caracterizando como um dos algoritmos mais estáveis de ordenação, independentemente do modelo de dados de entrada.

A Figura II-C2 apresenta o processo de equivalência entre a estrutura de dados Heap na forma de árvore e arranjo.

O algoritmo consiste na construção de um *max heap* (cuja principal propriedade consiste em que sempre o elemento raiz

é maior que seus filhos, para todas as sub-árvores possíveis), em seguida, aplica-se a troca do primeiro elemento pelo último para todos os elementos do conjunto (equivalente a remover todos os elementos do arranjo), e posteriormente, rearranjar a estrutura para cada elemento, desse modo é obtido um vetor ordenado de modo crescente ao termino do processo.

Complexidade:

- **Caso médio:** $O(nlogn)$, de modo invariante.

D. Algoritmos baseados em Intercalação

1) Merge Sort: Este algoritmo utiliza o princípio de divisão e conquista, de modo a quebrar um problema complexo em sub-problemas, até o caso base (restante um ou dois elementos no sub-vetor), de modo que seja possível ordená-los. A partir da garantia de que todos os sub-arranjos estão ordenadas, inicia-se o processo de combinação entre eles, de modo recursivo, até alcançar o problema original, de modo ordenado.

As posições situações de ocorrência são as seguintes:

- em seu seu melhor caso nunca é necessário realizar trocas após comparações;
- o caso médio ocorre quando nem sempre é necessário realizar trocas após comparações;
- por fim, o pior caso ocorre quando sempre é necessário realizar trocas após comparações.

Contudo, de modo invariável, todos os casos tem a mesma complexidade assintótica. Adicionalmente, vale-se destacar que diferentemente dos demais algoritmos que não se utilizam de memória adicional, o algoritmo Merge Sort tem complexidade espacial de tamanho n, que em dadas situações, deve ser levada em consideração.

A Figura II-D1 apresenta o processo de divisão e conquista utilizado no algoritmo.

Complexidade:

- **Caso médio:** $O(nlogn)$, de modo invariante.

III. Análise Experimental

De modo a prover uma análise experimental em relação aos algoritmos, foram conduzidos estudos com diferentes tamanhos de entrada, cujas dimensões foram: 10, 100, 1 000, 10 000, 100 000, 500 000 e 1 000 0000 de elementos, em que para todos os casos foram utilizados como teste um vetor crescente, decrescente e elementos dispersos pelo vetor de forma aleatória (vale ressaltar que para este último foi utilizado um arquivo que armazena os elementos, para que esta sequência aleatória seja sempre a mesma para execução em todos os algoritmos). Todos os casos foram avaliados em

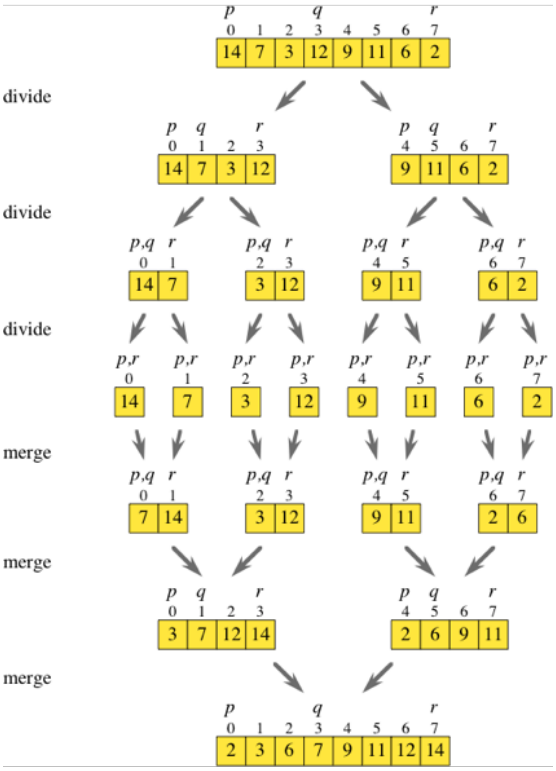

Fig. 4. Abordagem de divisão e conquista do algoritmo.

relação ao tempo medido em milissegundos (ms) e possuíram três execuções, em que foi escolhido o caso mediano.

Para a realização da análise experimental foi utilizado um notebook com as seguintes especificações: processador Intel Core i7-7700HQ CPU @ 2.80 GHz de 4 núcleos físicos e 8 threads, com memória cache L1 de 256 KB, L2 de 1 MB, L3 de 6 MB e memória RAM de 8 GB DDR4.

A. Modelos de entrada de dados

As análises experimentais a seguir realizam uma comparação entre todos os métodos de ordenação para cada caso de entrada de dados específico, sendo ordenado de modo crescente, decrescente e aleatório.

1) Entrada crescente: Inicialmente, através da Figura 5, pode-se observar que as estratégias de ordenação dos algoritmos *Bubble Sort Clássico*, *Selection Sort* e o *Quick Sort* com pivô inicial apresentaram desempenhos inferiores aos demais, principalmente em relação a esse último, que a partir de uma dada grandeza de entrada superou os outros dois, desse modo é possível notar o impacto que a quebra inadequada de partições pode ocasionar.

Em relação aos outros algoritmos, esses apresentaram melhores desempenhos, assemelhando-se a algoritmos lineares, devido a presença de estratégias que reduzem o número de operações sobre os elementos, com destaque ao *Insertion Sort*, um algoritmo inicialmente quadrático, torna-se o mais eficiente entre os analisados.

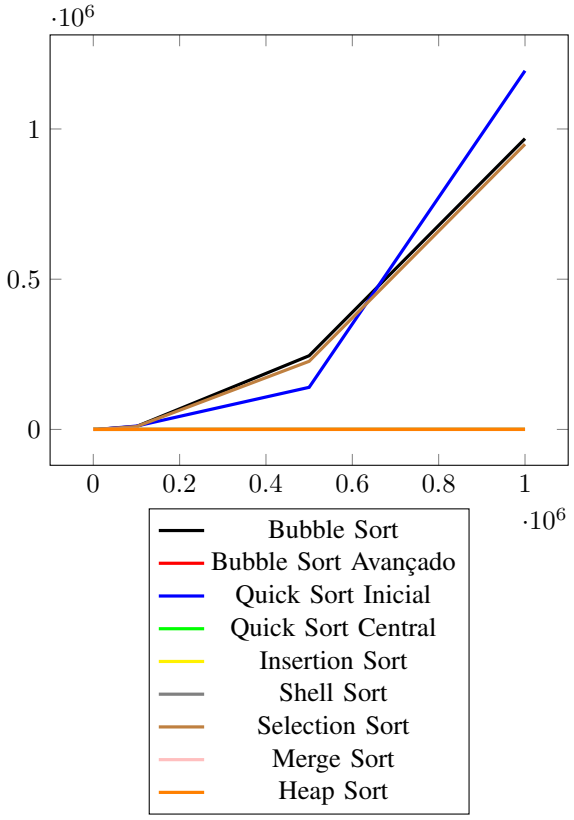

Fig. 5. Comparativo de execução para entrada crescente.

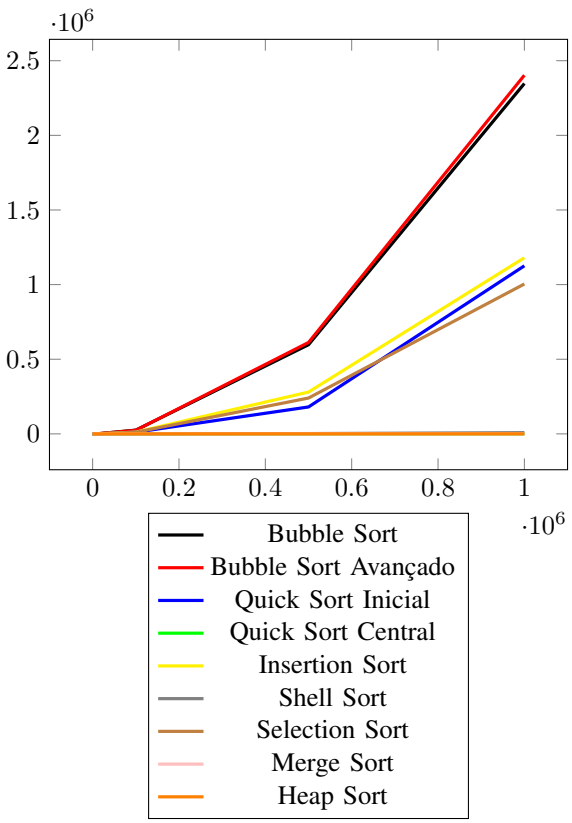

Fig. 6. Comparativo de execução para entrada decrescente.

2) Entrada decrescente: Através do gráfico representado pela Figura 6, permite-se observar a formação de três conjuntos de desempenho de algoritmos. No primeiro conjunto, pior desempenho, composto pelas duas implementações do *Bubble Sort*, é possível caracterizar o pior caso de ambos, de modo que nenhuma estratégia adicional para otimização tem efeito no desempenho.

Em seguida, no segundo conjunto, enquadram-se os algoritmos *Insertion Sort*, *Quick Sort* com pivô inicial e *Selection Sort*, com um comportamento próximo de quadrático, porém com um menor número de operações de comparação entre detrimento ao primeiro grupo.

O último caso dessa análise estão os algoritmos *Heap Sort*, *Merge Sort*, *Shell Sort* e *Quick Sort* com pivô central, demonstrando que em dadas situações um algoritmo baseado em inserção pode se equiparar a implementações mais sofisticadas.

3) Entrada aleatória: O experimento revelou que para esta organização do conjunto os algoritmos Heap Sort, Merge Sort, Shell e Quick Sort para ambas implementações tiveram um desempenho considerado superior com relação aos demais. Os algoritmos de pior rendimento consistem nas duas implementações do Bubble Sort.

No último modelo de entrada de dados, representado pela

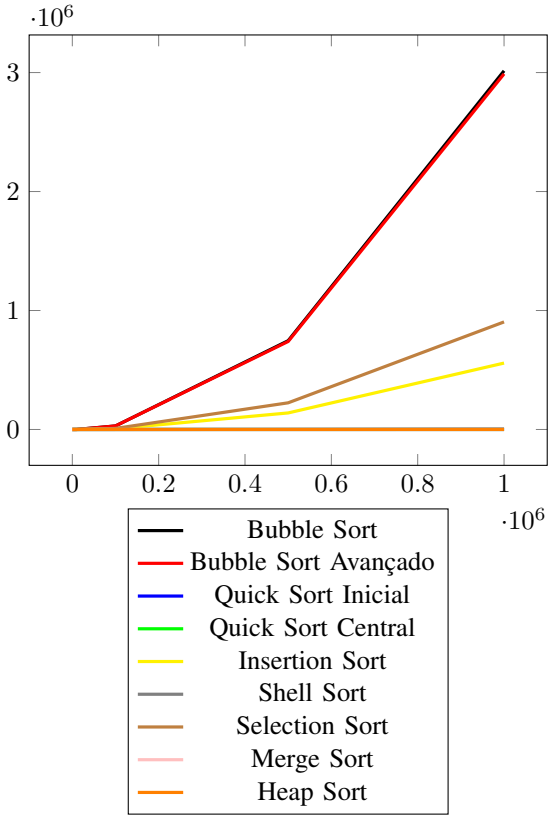

Fig. 7. Comparativo de execução para entrada aleatório.

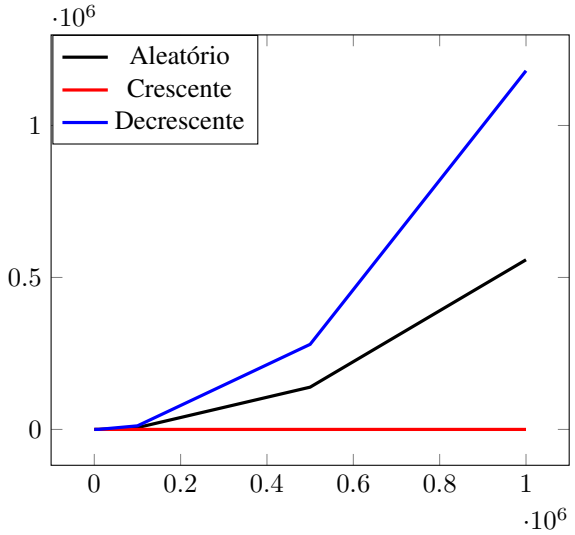

Fig. 8. Análise experimental do algoritmo Insertion Sort.

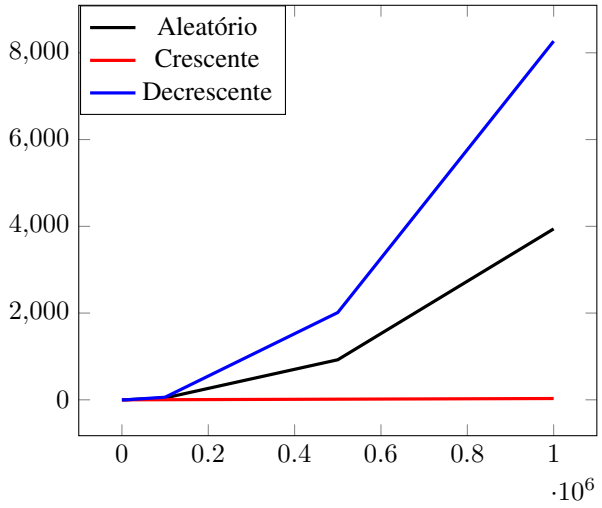

Fig. 9. Análise experimental do algoritmo Shell Sort.

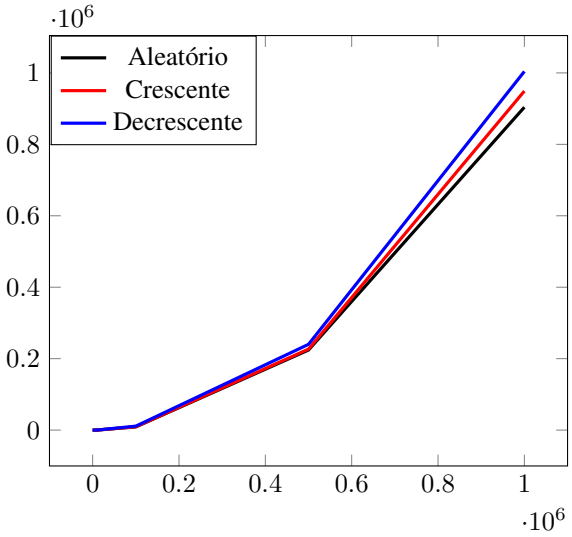

Fig. 10. Análise experimental do algoritmo Selection Sort.

Figura 7, é possível observar que os algoritmos *Heap Sort*, *Merge Sort*, *Shell Sort* e *Quick Sort* obtiveram um desempenho com relação aos demais. De modo intermediário, os algoritmos *Selection Sort* e *Insertion* tiveram um desempenho pior se comparado ao primeiro grupo.

Por fim, os algoritmos de pior rendimento consistem nas duas implementações do *Bubble Sort*, em que deve-se destacar o elevadíssimo número de comparações como principal fator para o rendimento apresentado.

B. Comparativo por algoritmo

1) Insertion Sort: Por meio da Figura 8, permite observar que o algoritmo Insertion Sort consiste em um excelente opção para arranjos quase ordenados, pois caracteriza-se com complexidade quase linear, podendo ser muito útil em cenários que haja a construção do conjunto ordenado por demanda, indicado pela curva do arranjo crescente, contudo para os demais casos este torna-se próximo da complexidade quadrática, não consistindo na melhor escolha para tais situações.

2) Shell Sort: O gráfico representado pela Figura 9 demostra análise para diferentes casos de entrada do algoritmo Shell Sort, o qual apresenta um comportamento semelhante ao anterior, porém é capaz de lidar com tamanhos de conjunto de dados em um tempo muito mais inferior, enquanto o primeiro trabalha na grandeza de milhões de unidades de tempo, no caso milissegundos, o segundo trabalha na proporção dos milhares.

Como a sua implementação consiste em uma melhoria da abordagem de inserção, a mesmas indicações de uso valem para este algoritmo.

3) Selection Sort: Através do gráfico representado pela Figura 10, pode-se concluir que o algoritmo de ordenação Selection Sort, apesar de ser indiferente quanto a organização prévia dos elementos pertencentes ao arranjo e ter uma das mais simples implementações, caracterizando-se como força-bruta, torna-se pior a medida que o número de elementos cresce, cuja curva se assemelha às quadráticas.

O principal e único destaque do algoritmo está relacionado ao número de trocas realizadas representado pelo gráfico na Figura !!!!!!!!!!!!GRÁFICO DE TROCAS!!!!!!!!!!!!!!!

4) Heap Sort: Por fim, o gráfico acima demonstra que o algoritmo Heap Sort mediante a uma implementação sofisticada promove resultados invariantes para todos os casos, sem o uso adicional de recursos.

5) Bubble Sort: A análise experimental realizada com o método Bubble Sort em sua versão clássica revelou que independente do caso em que os elementos estão distribuídos no vetor, seja de modo aleatório, crescente ou decrescente o comportamento expresso no gráfico acima é semelhante, considerado de ordem quadrática, tendo para a organização inicial um vetor crescente seu desempenho foi levemente superior aos demais devido ao número reduzido de trocas entre elementos.

Os resultados da análise experimental revelaram que para

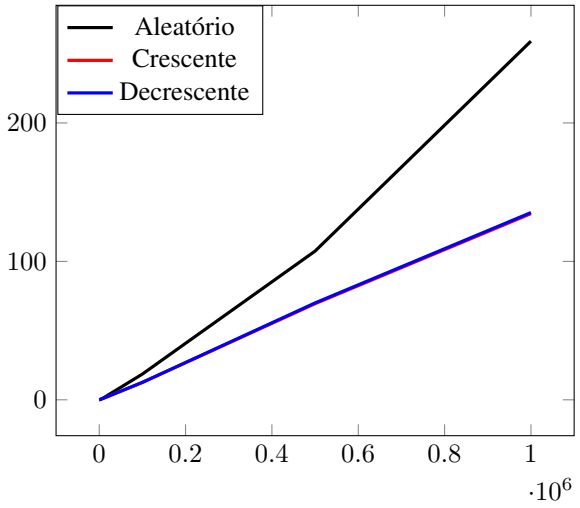

Fig. 11. Análise experimental do algoritmo Heap Sort.

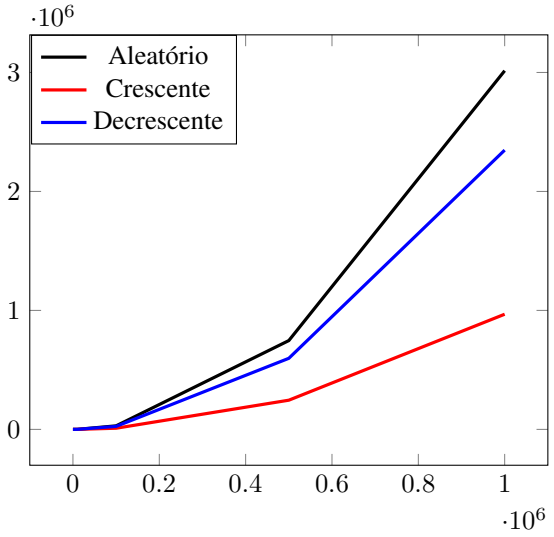

Fig. 12. Análise experimental do algoritmo Bubble Sort.

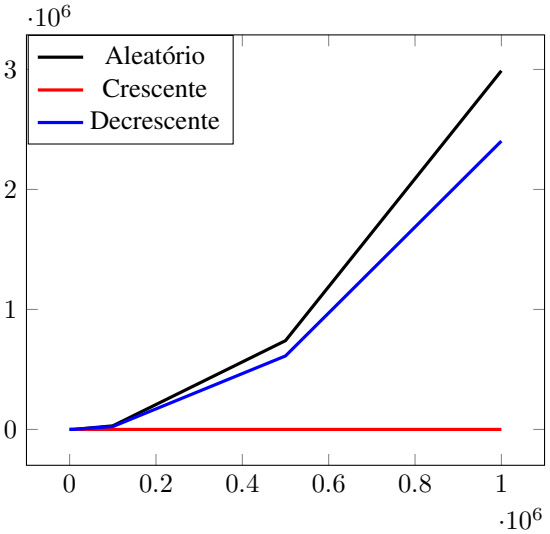

Fig. 13. Análise experimental do algoritmo Bubble Sort Avançado.

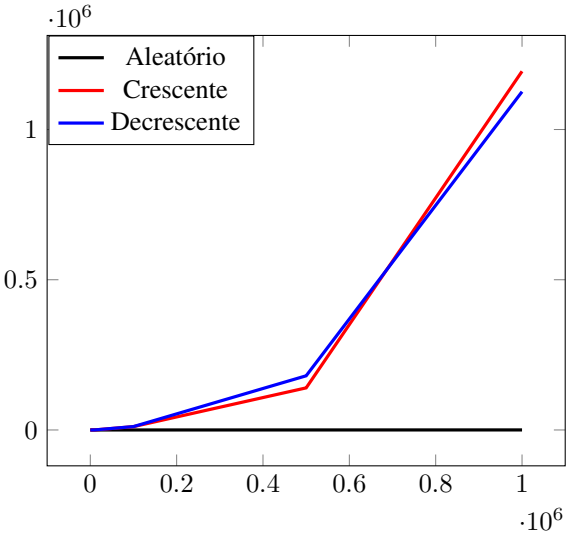

Fig. 14. Análise experimental do algoritmo Quick Sort com pivô inicial.

o método de ordenação Bubble Sort com melhorias apresenta como seu melhor caso quando o vetor já está organizado de modo crescente tendo um desempenho linear, além disso tanto para conjuntos organizados de modo decrescente quanto para conjuntos organizados de forma aleatória temos um comportamento que se enquadra em seu pior caso com desempenho considerado quadrático, que é visivelmente observado em conjuntos de tamanho maior que cem mil.

6) Quick Sort: O experimento demonstrou que o algoritmo Quick Sort com pivô inicial tem um bom desempenho para um conjunto organizado em ordem aleatória, contudo para conjuntos em ordem crescente e decrescente o teste resultou em um baixo desempenho aumentando de forma considerável o tempo de execução.

Considerando a implementação do algoritmo Quick Sort utilizando um o pivô central é notório que nos casos em que o vetor está inicialmente ordenado de modo crescente e de modo decrescente o desempenho são semelhantes e melhores se comparados com um vetor com elementos dispostos aleatoriamente.

7) Merge Sort: Em suma, a sua implementação baseada em divisão e conquista, permite um remanejamento dos elementos quase indiferente para os casos de conjunto crescente, decrescente e aleatório, colocando como um dos melhores algoritmos, apesar de seu maior consumo de memória.

C. Comparativo de abordagem

1) Abordagem de inserção:
2) Abordagem de troca:
3) Abordagem de seleção:

IV. Considerações Finais

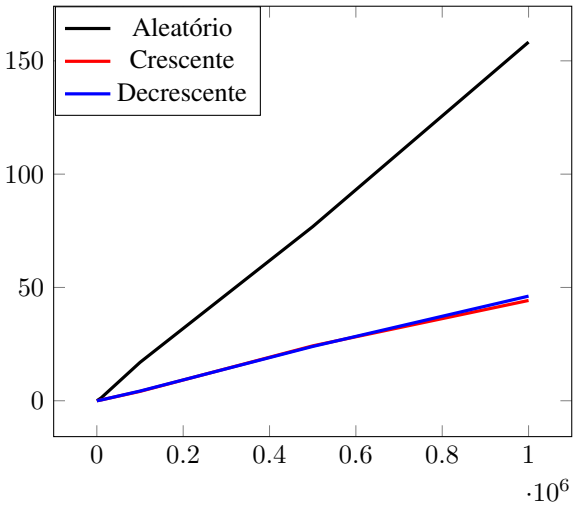

Fig. 15. Análise experimental do algoritmo Quick Sort com pivô central.

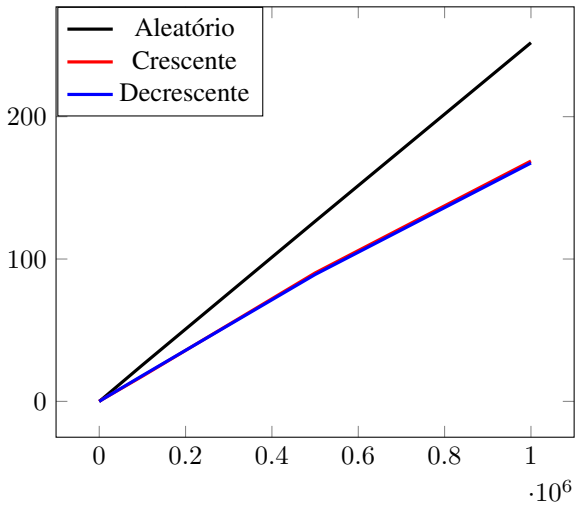

Fig. 16. Análise experimental do algoritmo Merge Sort.