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 apresentadas as análises experimental aplicadas em conjuntos de entradas com configurações variadas. A Seção 4 apresenta brevemente uma comparação do número de trocas que cada algoritmo realiza para ordenar cada conjunto de entrada. Por fim, na Seção 5, 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 [2].

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 [2].

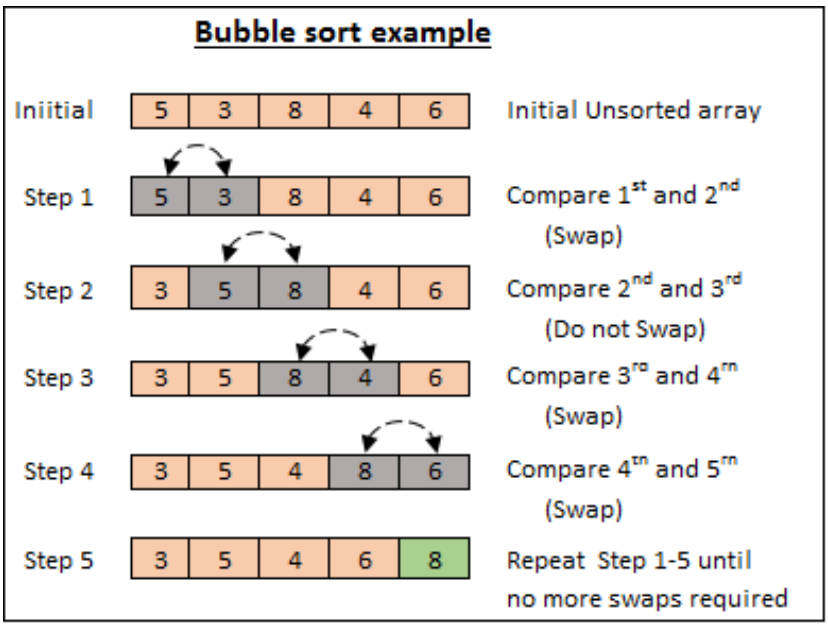

Fig. 1. Modelo de comparação de elementos adjacentes.

A Figura 1 apresenta a abordagem de ordenação utilizada pelo algoritmo referido.

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 [1].

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 [1].

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 [2].

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 [1].

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 [1]. 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.

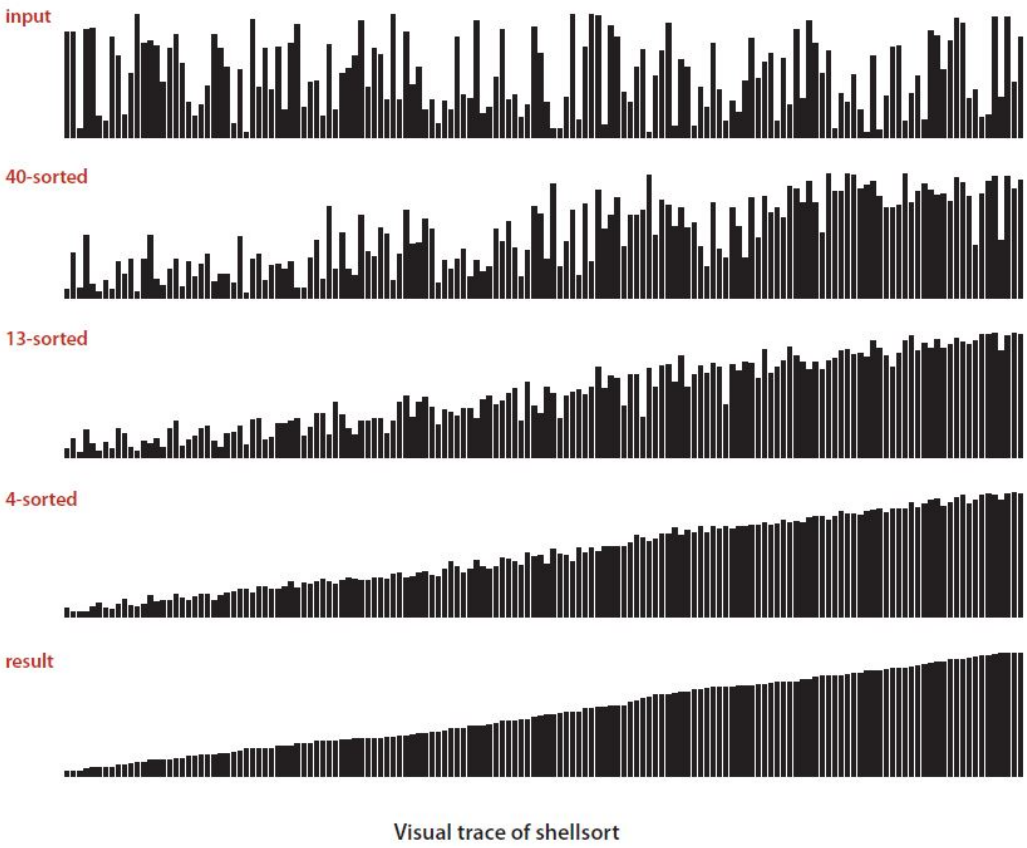

Fig. 2. Modelo de partições com elementos distantes em diferentes tamanhos.

A Figura 2 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 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 [1][2].

O grande destaque para esse algoritmo, dentro do contexto das abordagens quadráticas, consiste no reduzido número de trocas, que no máximo ocorre n vezes [2].

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 [1].

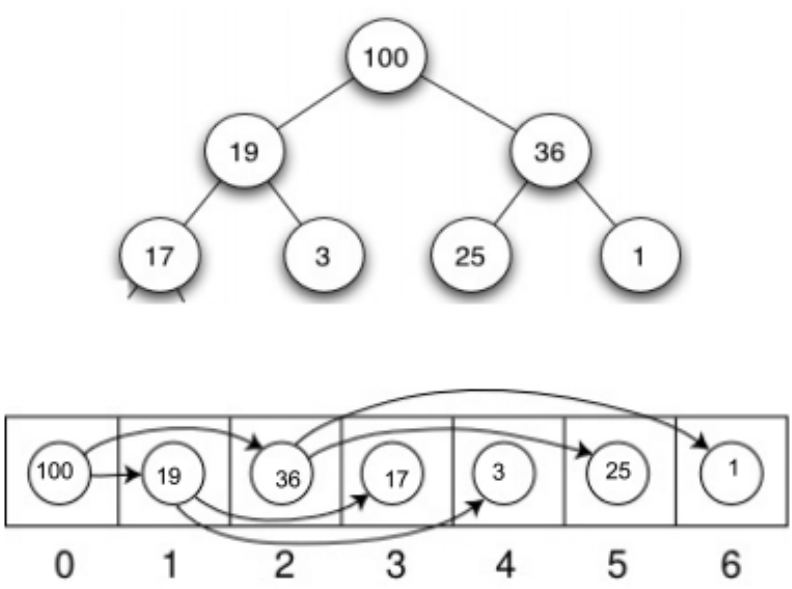

Fig. 3. Equivalência entre a árvore e arranjo.

A Figura 3 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 [1].

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 [3].

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 [3].

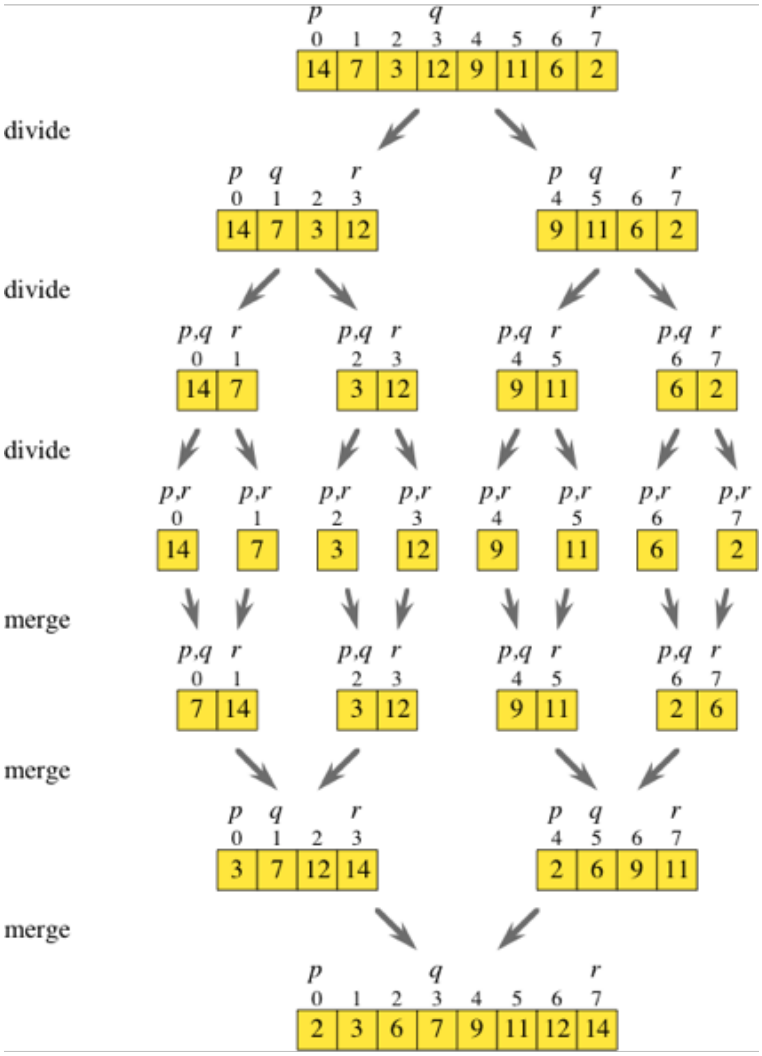

Fig. 4. Abordagem de divisão e conquista do algoritmo.

A Figura 4 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 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.

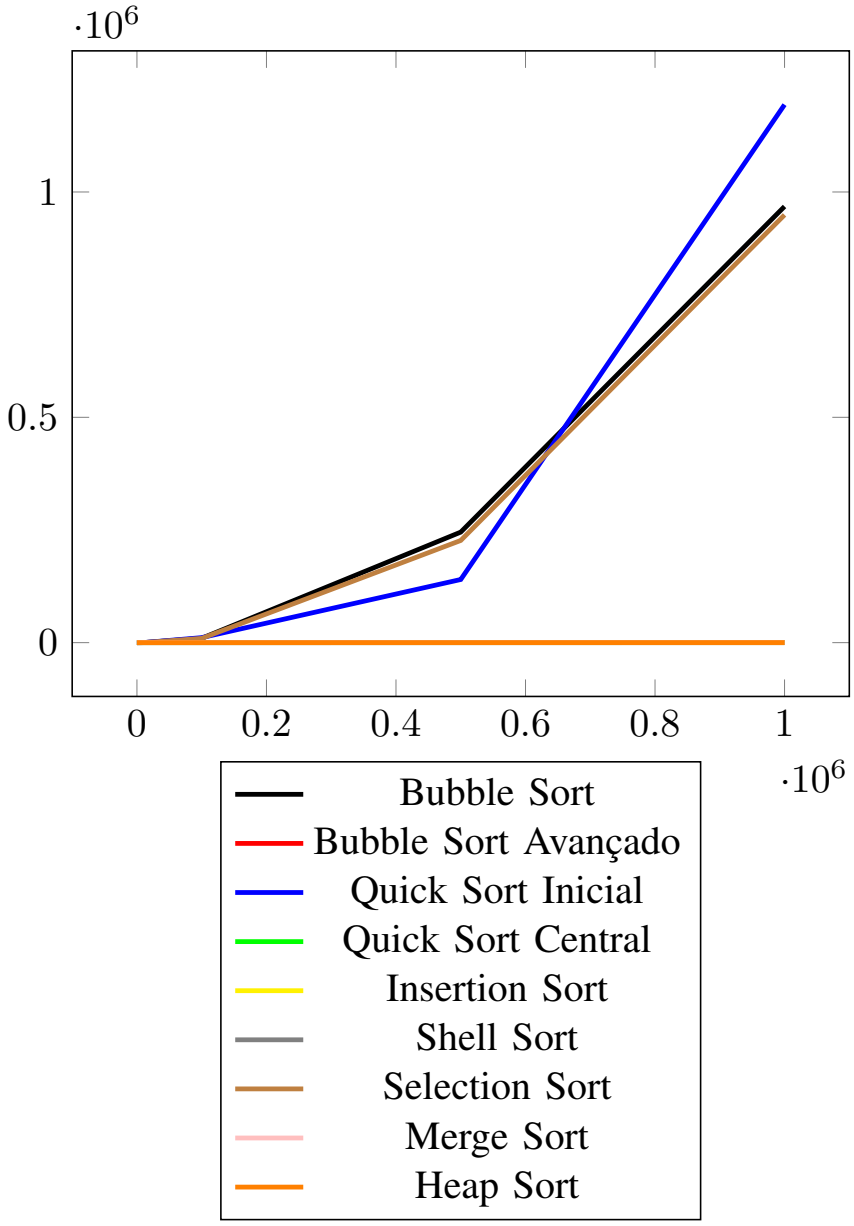

Fig. 5. Comparativo de execução para entrada crescente.

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.

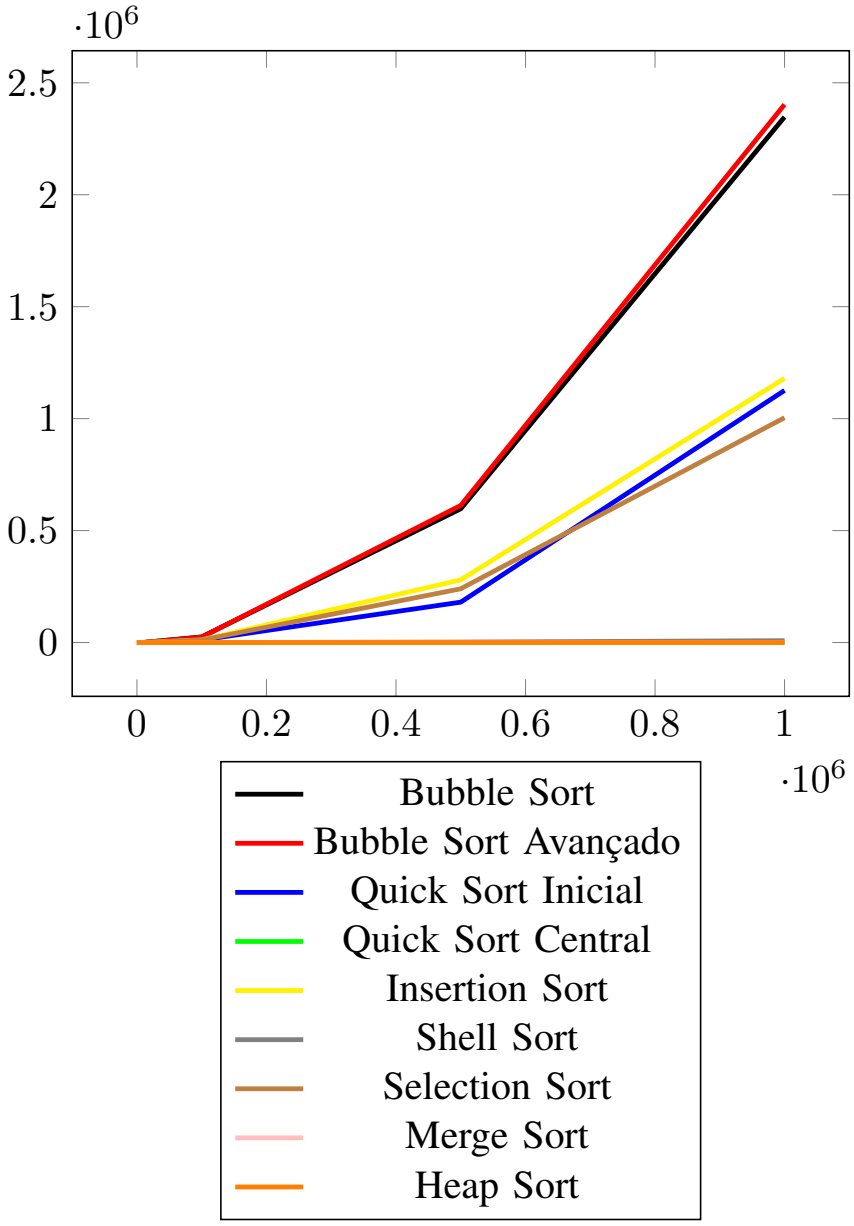

Fig. 6. Comparativo de execução para entrada decrescente.

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.

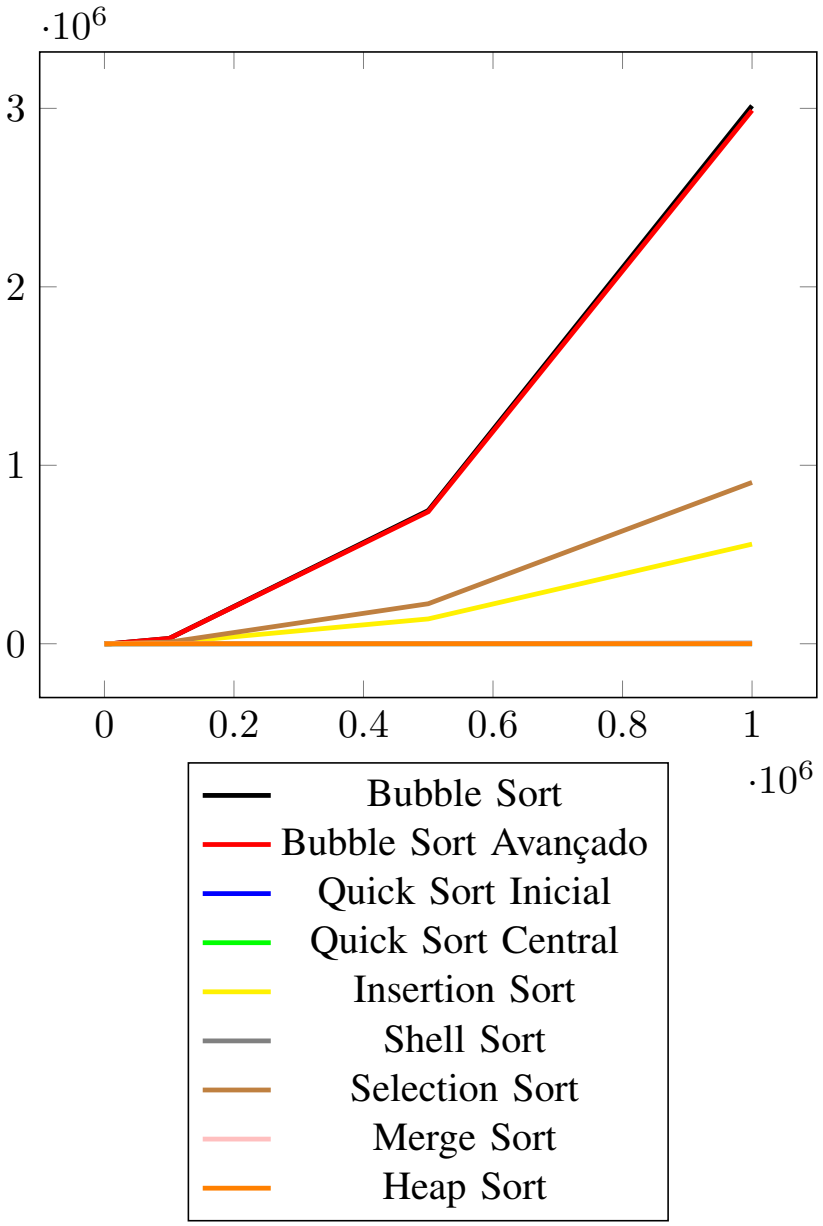

Fig. 7. Comparativo de execução para entrada aleatório.

No último modelo de entrada de dados, representado pela 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.

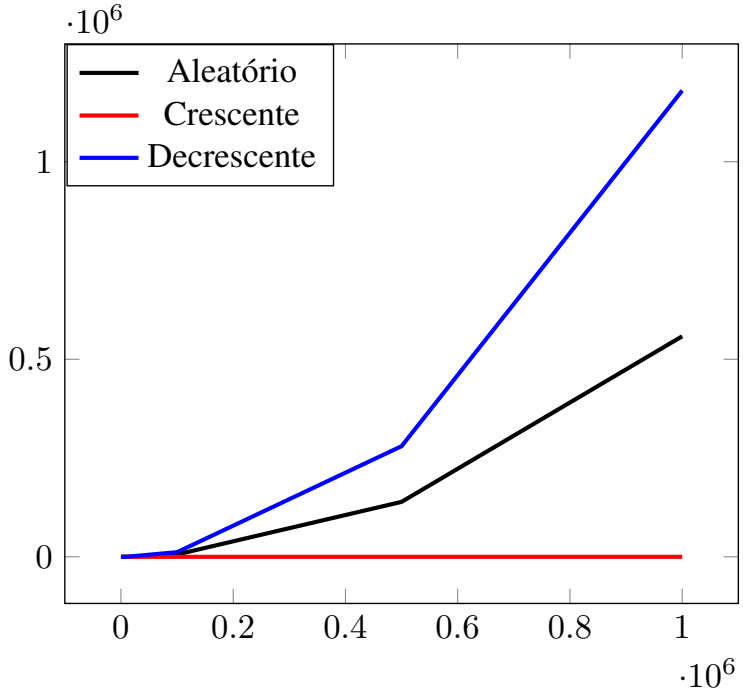

Fig. 8. Análise experimental do algoritmo Insertion Sort.

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.

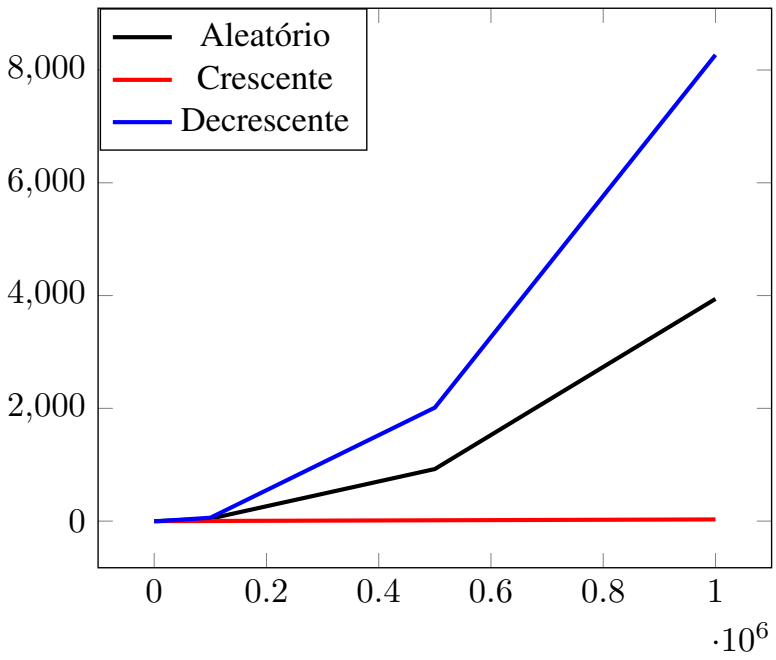

Fig. 9. Análise experimental do algoritmo Shell Sort.

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.

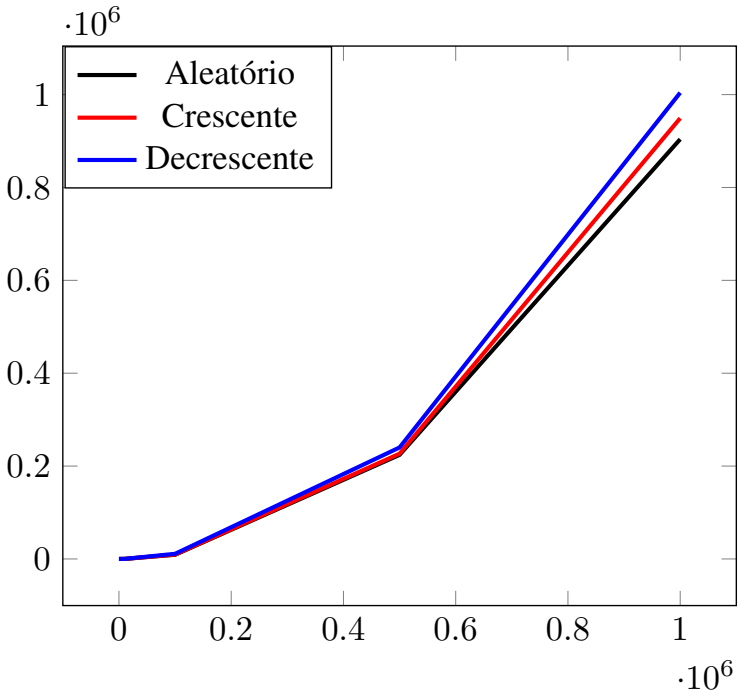

Fig. 10. Análise experimental do algoritmo Selection Sort.

4) Heap Sort: Para a análise do algoritmo de ordenação *Heap Sort* é possível observar, através do gráfico representado pela Figura 11 que os melhores casos são para entradas crescentes e decrescentes, enquanto que para o caso aleatório é registrado uma complexidade de tempo.

Como grande destaque, de modo invariante para entrada de dados, este consiste o único algoritmo que realizou todos os testes experimentais dentro da grandeza de centenas de millissegundos, não alcançando os 300 ms para as maiores entradas de dados.

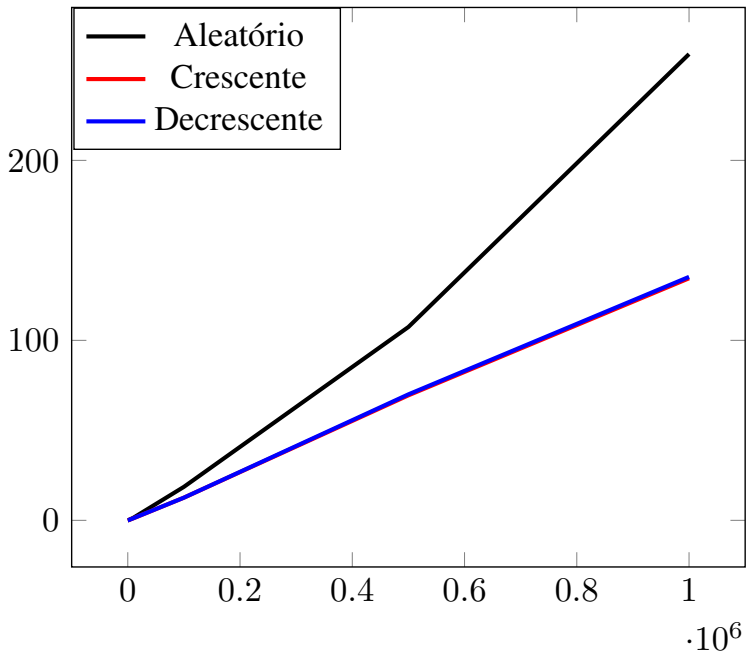

Fig. 11. Análise experimental do algoritmo Heap Sort.

5) Bubble Sort: O estudo experimental, apresentado na Figura 12 realizado com o método de ordenação *Bubble Sort* original, demonstra que independentemente da entrada de dados, o comportamento é semelhante, considerado de ordem quadrática, em que, de modo adicional, para um vetor crescente seu desempenho foi levemente superior aos demais devido ao número reduzido de trocas entre elementos.

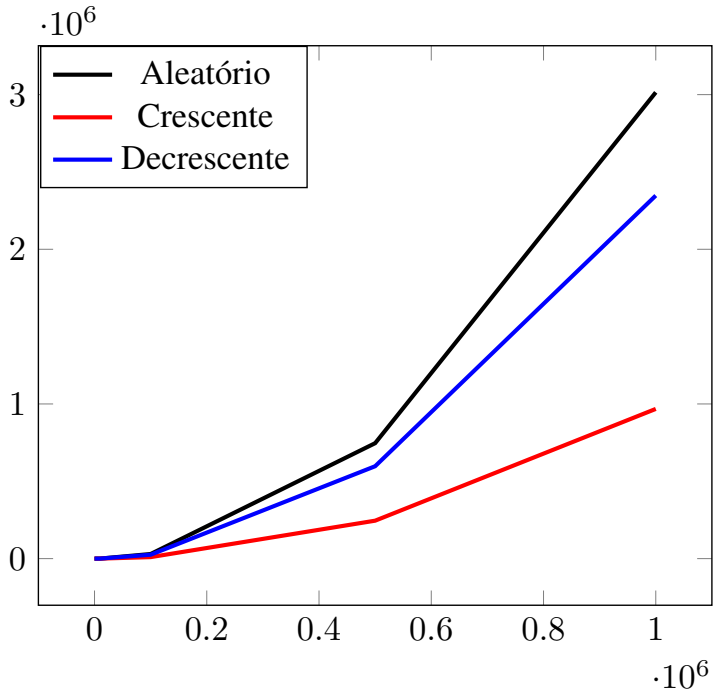

Fig. 12. Análise experimental do algoritmo Bubble Sort.

Em contraponto, a implementação com com melhorias do algoritmo *Bubble Sort*, representado pela Figura 13, apresenta como melhor caso o arranjo crescente, registrando um desempenho linear. Para os demais casos de entrada, o comportamento

quadrático novamente é registrado, e torna-se mais evidente de modo proporcional ao crescimento do conjunto de entrada.

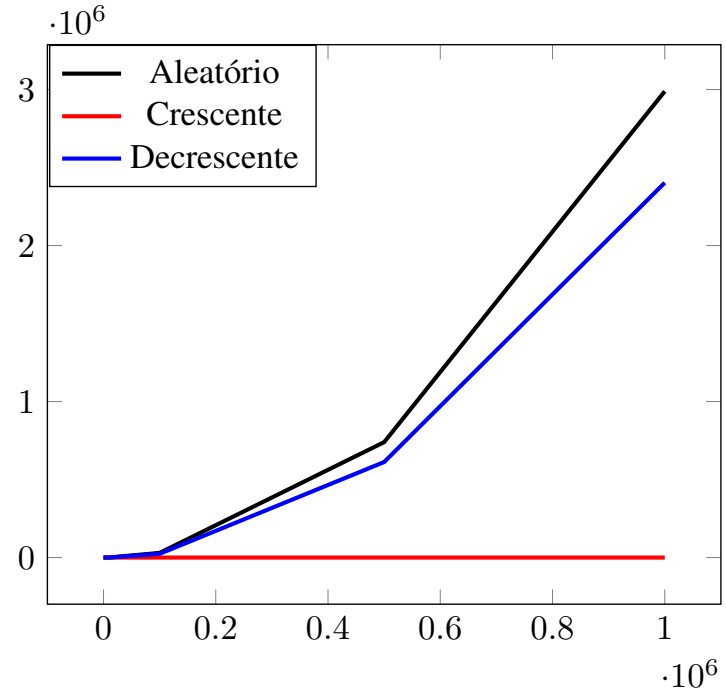

Fig. 13. Análise experimental do algoritmo Bubble Sort Avançado.

6) Quick Sort: A análise experimental realizada para o algoritmo *Quick Sort* com pivô inicial, representada pela Figura 14 apresentou bom desempenho para um conjunto aleatório, contudo para ordem crescente e decrescente, o processo de teste determinou desempenho inferior de forma considerável.

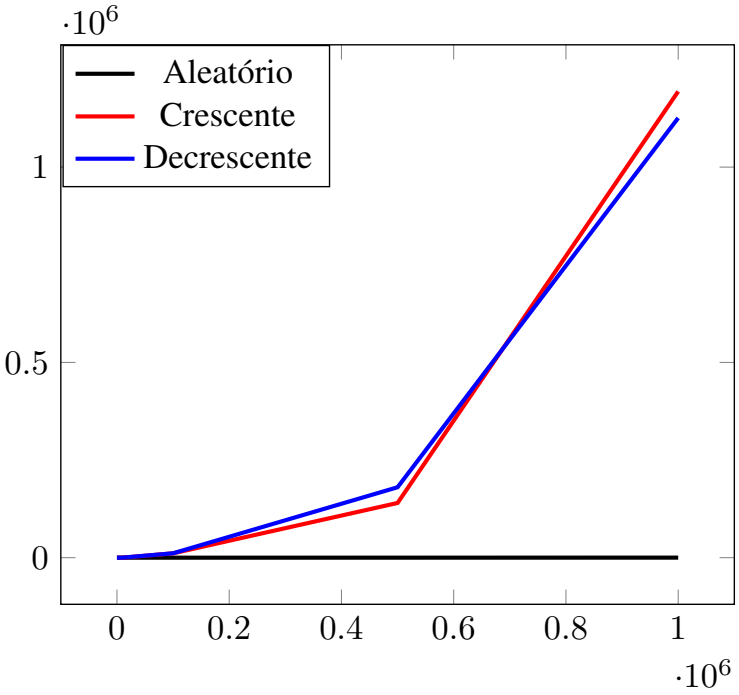

Fig. 14. Análise experimental do algoritmo Quick Sort com pivô inicial.

Por outro lado, considerando a implementação do algoritmo *Quick Sort* utilizando pivô central, é notório que nos casos em que o vetor está inicialmente ordenado de modo crescente e decrescente o desempenho é superior em detrimento ao arranjo disposto aleatoriamente.

Vale-se destacar a disparidade na grandeza de tempo entre a abordagem de pivô e central. Em ambos os casos, para a maior entrada de dados, a primeira opção requereu tempo superior a 1 000 000 ms, enquanto o segundo requereu um pouco mais que 150 ms.

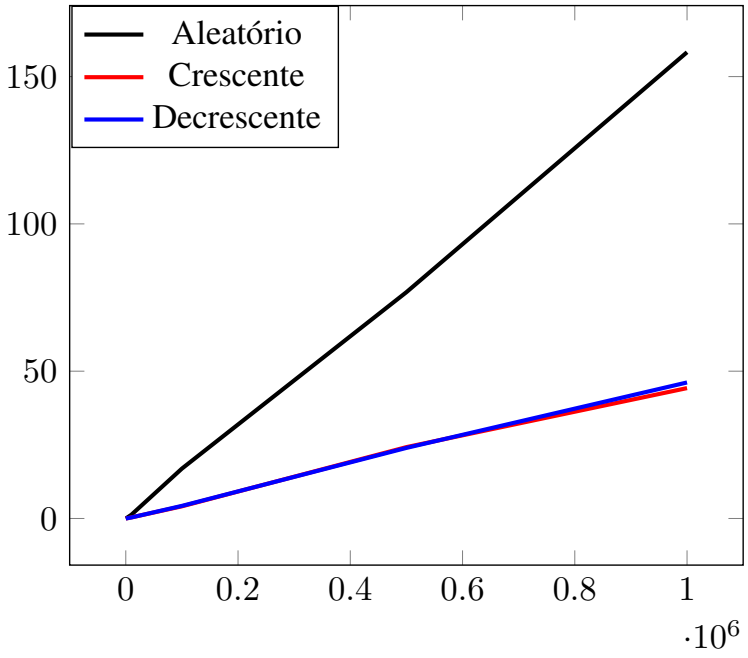

Fig. 15. Análise experimental do algoritmo Quick Sort com pivô central.

7) Merge Sort: Por fim, para o algoritmo *Merge Sort* que possui implementação baseada em divisão e conquista, a análise representada pela Figura 16, permite o processo de ordenação de modo quase indiferente em relação ao modelo de entrada de dados. De modo específico, é possível observar uma pequena vantagem para conjunto decrescentes e crescentes em detrimento ao aleatório.

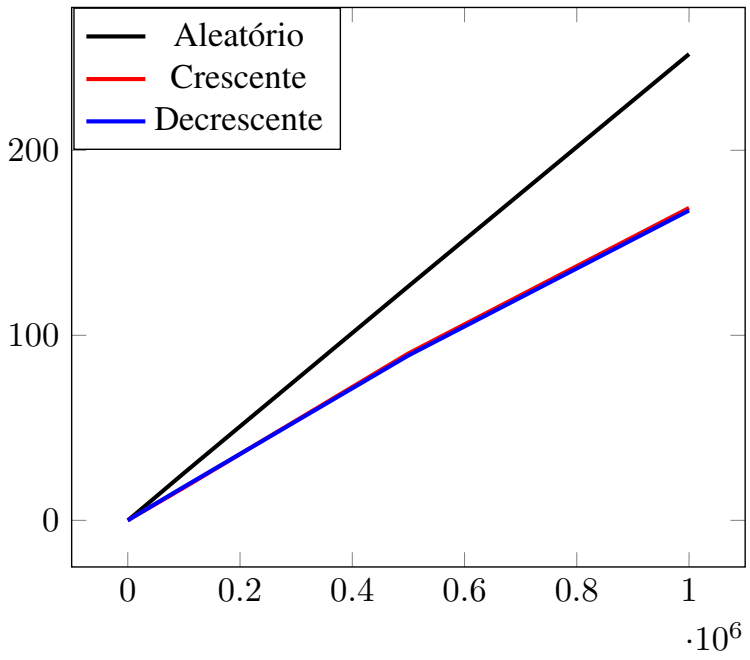

Fig. 16. Análise experimental do algoritmo Merge Sort.

C. *Comparativo de abordagem*

Todas as comparações abaixo foram realizadas utilizando os casos para conjunto de dados aleatório e também com as melhores implementações de cada algoritmo, de forma a não favorecer determinados algoritmos e consequentemente propor conclusões equivocadas.

1) Abordagem de inserção: A comparação entre os algoritmos com abordagem de ordenação baseada em inserção, representado pela Figura 17 indica uma enorme superioridade para o algoritmo *Shell Sort* em detrimento ao *Insertion Sort*, a qual se eleva ainda mais após entradas de tamanho 500 000, demonstrando que uso de saltos para ordenação em sub-partições é essencial, proporcionando grande ganho de desempenho.

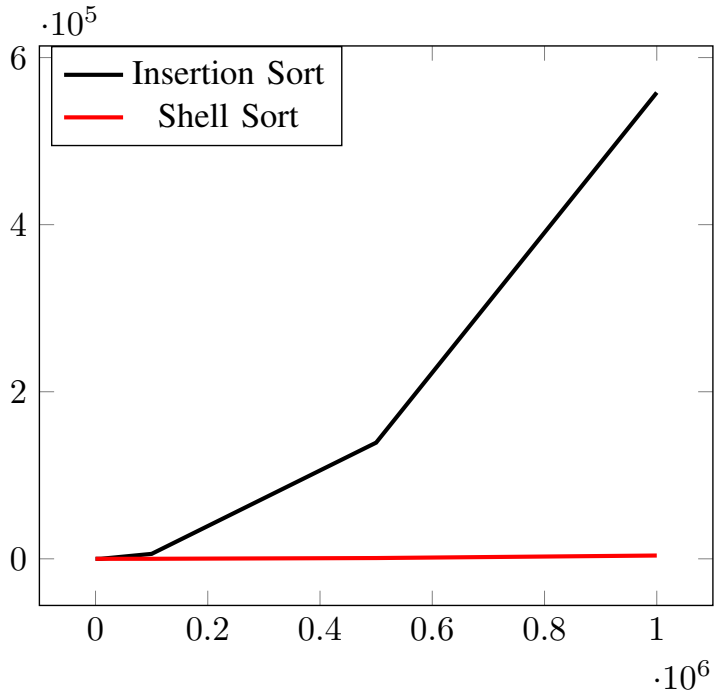

Fig. 17. Comparativo de algoritmos baseados em inserção.

2) Abordagem de troca: Para o comparativo dos algoritmos de ordenação baseados em troca, através da Figura 18, pode-se observar que o algoritmo *Quick Sort* com pivô central tem custo de processamento extremamente inferior em relação ao *Bubble Sort* com melhorias. Desse modo, vale ressaltar que a ordenação através de sub-partições e a escolha adequada de um pivô proporcionam um resultado muito superior do que a comparação direta entre vizinhos usada no *Bubble Sort*.

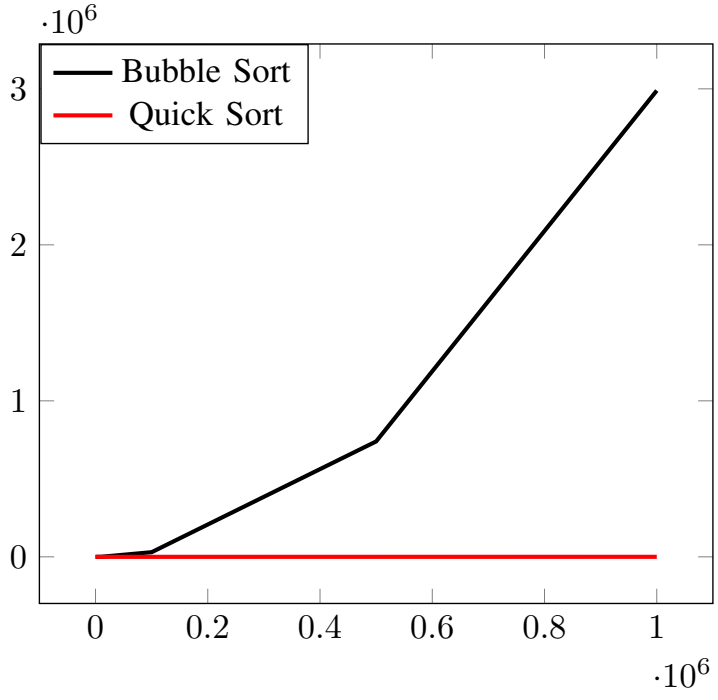

Fig. 18. Comparativo de algoritmos baseados em troca.

3) Abordagem de seleção: Por fim, o comparativo da abordagem de ordenação baseada em seleção, representada pela Figura 19, indique um desempenho extremamente superior do algoritmo *Heap Sort*, que utiliza a ideia de fila de prioridades, em detrimento ao algoritmo *Selection Sort*, que utiliza uma ideia intuitiva, porém caracterizada como força-bruta, sem qualquer inteligência do método ou aproveitamento de propriedades do conjunto de dados.

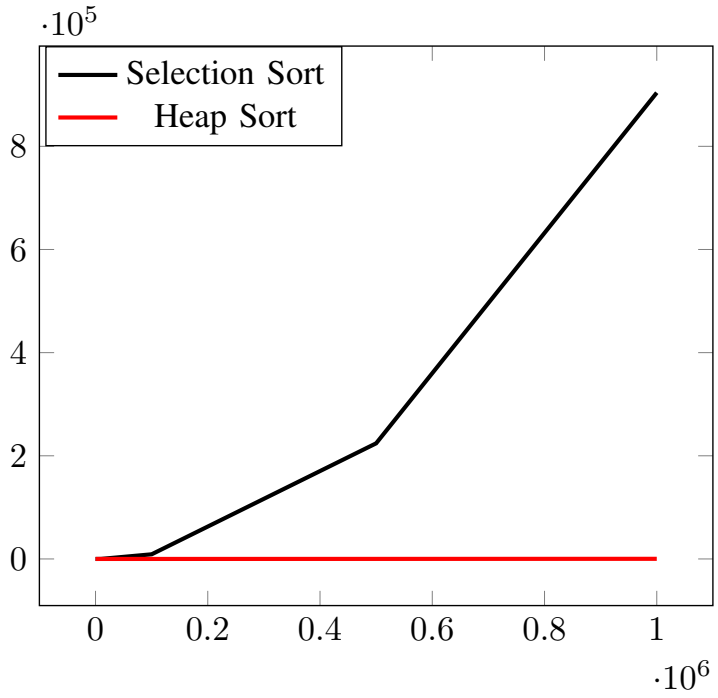

Fig. 19. Comparativo de algoritmos baseados em seleção.

IV. Análise quantidade de trocas realizadas

Além da análise experimental que visa comparar os tempos de execução entre os diversos algoritmos de ordenação, também foi aplicada uma análise com objetivo de comparar a quantidade de trocas de posições de elementos durante uma ordenação. Foi medido durante a execução dos algoritmos a quantidade máxima de troca que eles realizam de acordo com os três tipos de entrada (aleatória, crescente e decrescente) em diferentes dimensões.

1) Entrada aleatória: A Figura 20 apresenta um gráfico que contem o número de trocas de todos os algoritmos. Nesta figura é possivel observar em amarelo representado pelo Insertion Sort que realiza um maior número de trocas, existe uma sobreposição na linha em amarelo, pois o Bubble Sort e o Bubble Sort também tem exatamente o mesmo comportamento que o Insertion Sort, além desses o Shell Sort também realiza um número grande de trocas. Os demais algoritmos não são possíveis de ser observados nesta imagem, por terem número de trocas bem menores, assim eles foram colocados isoladamente na Figura 21.

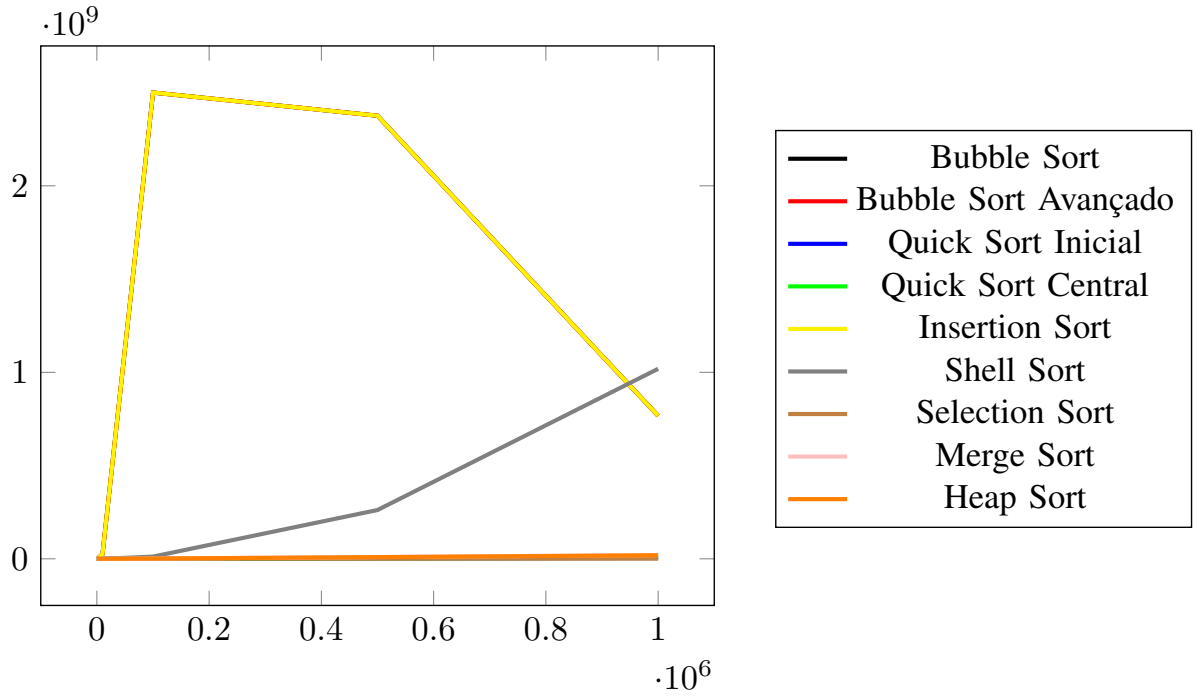

Fig. 20. Número de trocas realizadas pelos algoritmos para entrada com elementos em ordem aleatória.

Na Figura 21 é possível observar os algoritmos que realizam menores números de trocas em entrada aleatória. Dentre eles o Heap Sort e o Merge Sort realizam mais trocas, seguidos das duas implementações do Quick Sort e é notório também que o algoritmo Selection Sort realiza menos trocas em conjuntos de entrada aleatória. Todos os algoritmos para essa situação tem um comportamento linear.

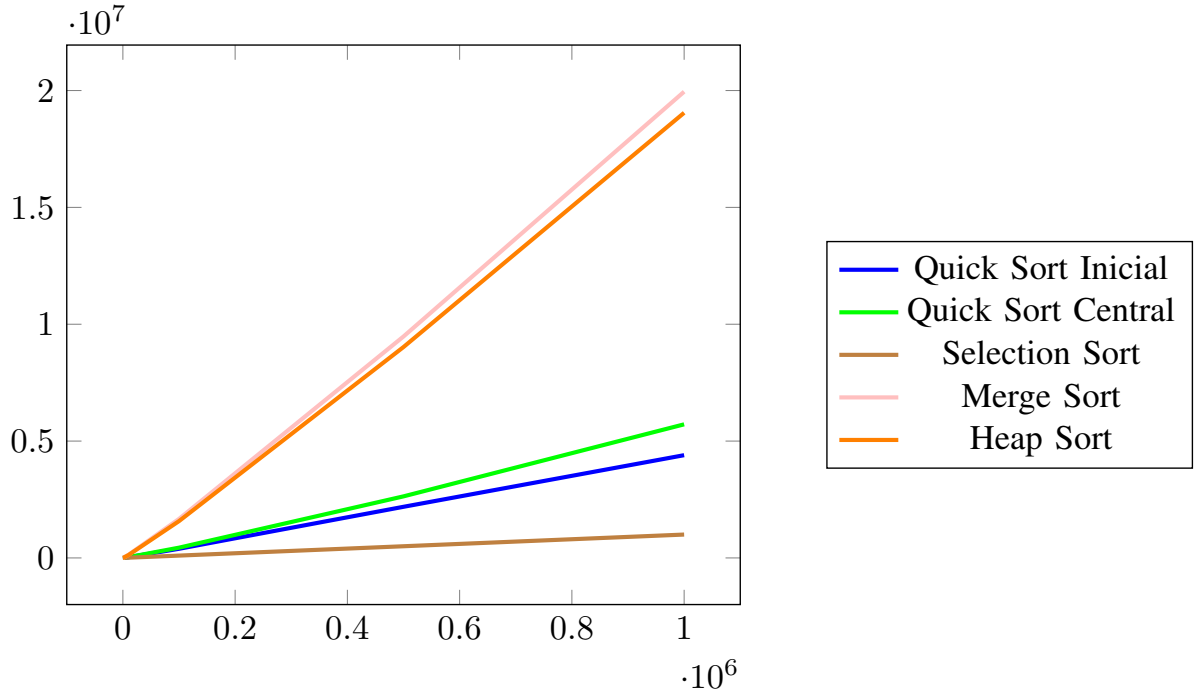

Fig. 21. Número de trocas realizadas pelos algoritmos para entrada com elementos em ordem aleatória.

2) Entrada crescente: A Figura 22 apresenta um gráfico com o comportamento em relação ao número de trocas para uma entrada crescente. Contudo visualiza-los todos juntos causa uma certa sobreposição. Para melhorar a visualização das análises

foram exibidos juntos os algoritmos que mais fazem trocas, os que menos fazem trocas e os que fazem trocas em valores intermediários com relação aos que mais fazem ou que menos fazem.

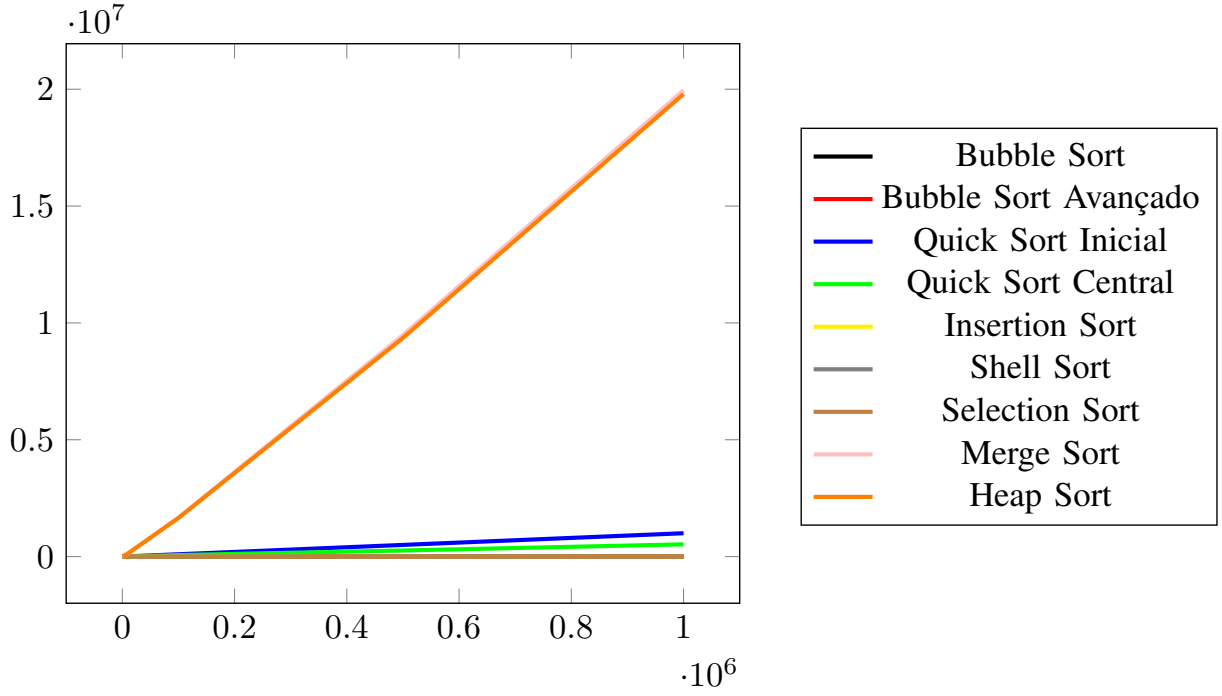

Fig. 22. Número de trocas realizadas pelos algoritmos para entrada com elementos em ordem crescente.

A Figura 23 apresenta a evolução do número de trocas dos algoritmos que mais fazem trocas, sendo eles Heap Sort e Merge Sort. O número de trocas que ambos fazem é bem semelhante, contudo o Merge Sort realiza um pouco mais trocas que o Heap Sort.

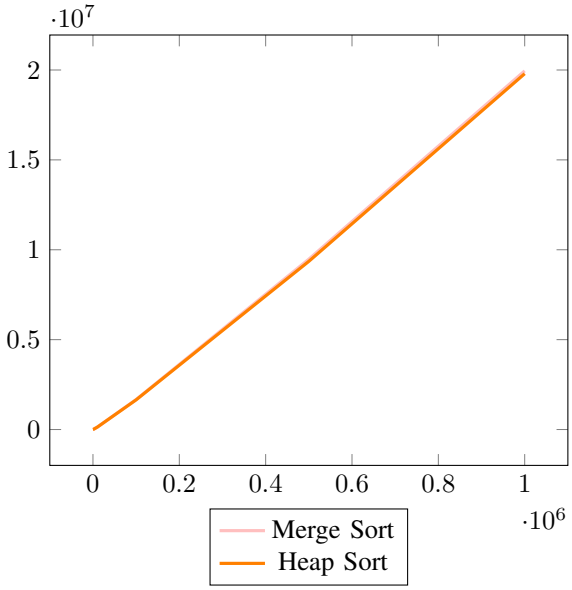

Fig. 23. Algoritmos que mais realizam trocas para entrada com elementos em ordem crescente.

A Figura 24 apresenta os algoritmos que menos realizam trocas para entrada crescente. No caso, eles não realizam nenhuma troca, pois seu principio de implementação verifica a necessidade ou não de realizar trocas, como o conjunto já está ordenado não é necessário realizar trocas.

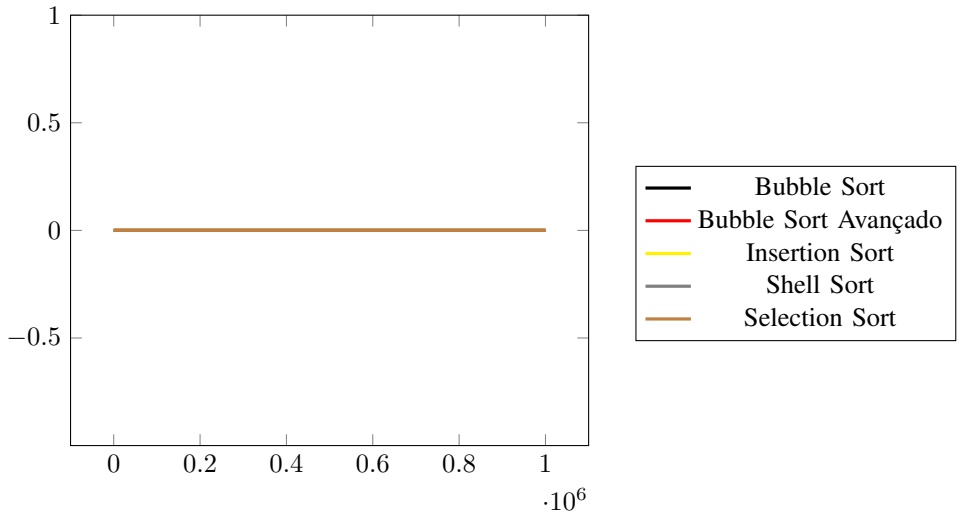

Fig. 24. Algoritmos que menos realizam trocas para entrada com elementos em ordem crescente..

Por fim, a Figura 25 apresenta os algoritmos que não fazem nem o maior nem o menor número de trocas, sendo eles as implementações do Quick Sort. Quando o Quick Sort é implementado de modo que o pivô é central ele faz menos trocas se comparado com o Quick Sort de implementação de pivô inicial.

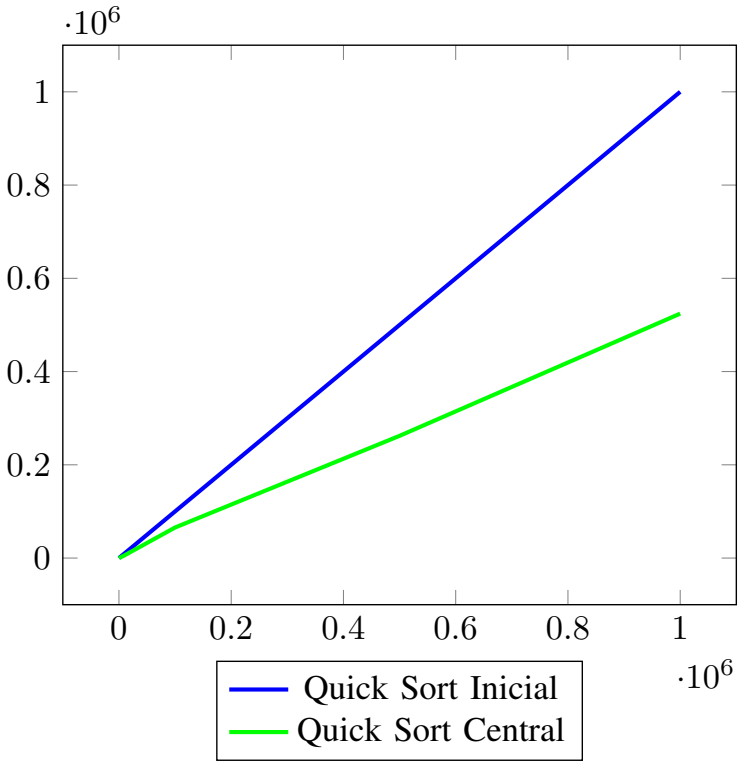

Fig. 25. Algoritmos que realizam um número mediano de trocas para entrada com elementos em ordem crescente..

3) Entrada decrescente: Para entradas com elementos dispostos em ordem decrescente os algoritmos que mais fazem trocas para ordenar o conjunto são o Bubble Sort (ambas implementações) e o Insertion Sort, seguidos pelo Shell Sort, observe a Figura 26

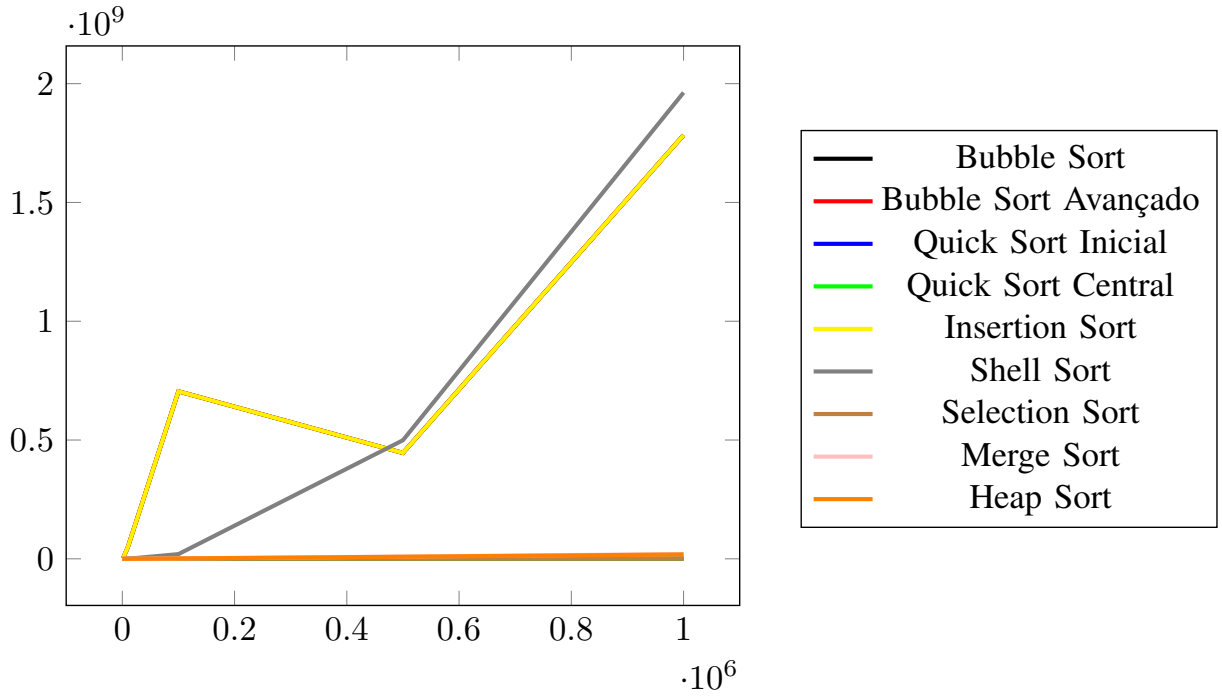

Fig. 26. Número de trocas realizadas pelos algoritmos para entrada com elementos em ordem decrescente.

Na Figura anterior não foi possível observar com exatidão o numero de trocas dos demais algoritmos, então eles foram agrupados na Figura 27 que ilustra a evolução do número de trocas de cada um deles. O algoritmo que menos faz troca para esta configuração de entrada é o Selection Sort, seguido pelas implementações do Quick Sort. Dentre esses algoritmos o Merge Sort e Heap Sort fazem mais bem mais trocas, porem ainda não fazem mais trocas que o Bubble Sort e Insertion Sort. O Merge Sort realiza um pouco mais de trocas que o Heap Sort.

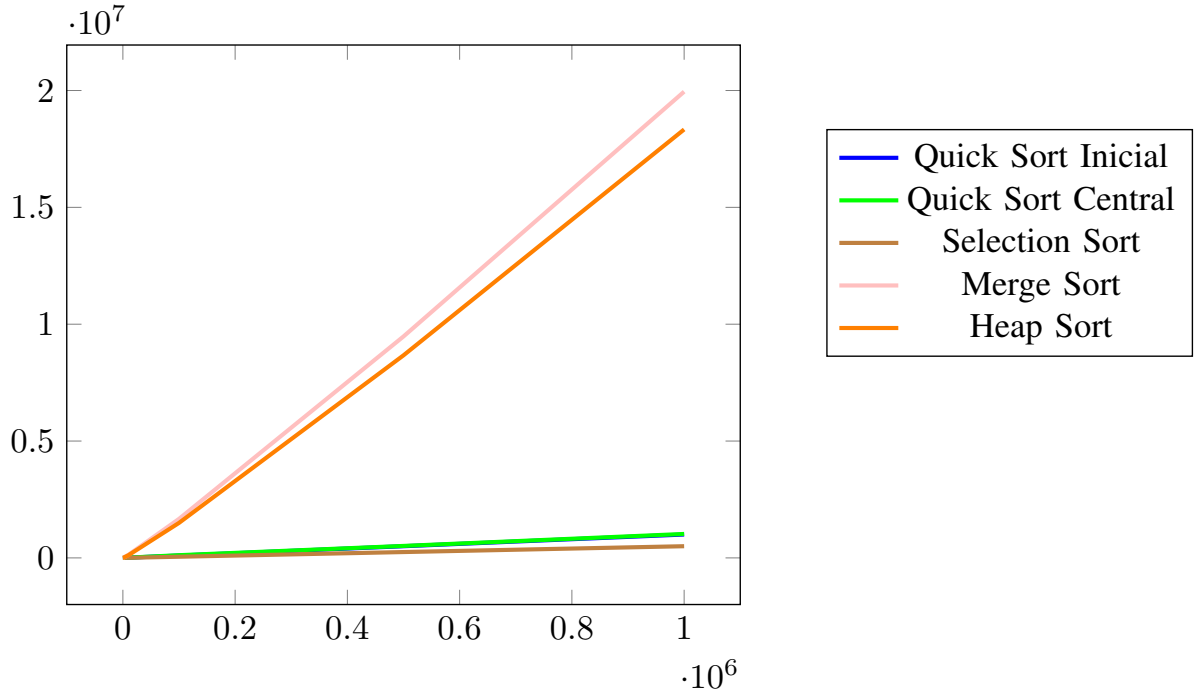

Fig. 27. Número de trocas realizadas pelos algoritmos para entrada com elementos em ordem decrescente.

A. Análise número de trocas por algoritmo

Após observar o comportamento do número de trocas com uma visão de todos os algoritmos juntos, agora é exposto esse comportamento considerando cada algoritmo individualmente, afim de perceber as diferenças existentes nos três tipos distintos de entradas (aleatório, crescente e decrescente).

1) Bubble Sort e Insertion Sort: Considerando o número de trocas necessário para ordenar um conjunto de dados, após a análise realizada foi constatado que os algoritmos implementados: Bubble Sort (implementação clássica e melhorada) e Insertion Sort, tiveram o mesmo comportamento, ou seja, o mesmo número de trocas. Tal comportamento pode ser observado na Figura 28.

Para a entrada crescente nenhuma troca foi realizada como era esperado, contudo para a entrada organizada em ordem decrescente tem-se um grande número de trocas para ordenar conjuntos maiores que 500000. A entrada aleatória é a que mais realiza trocas em conjuntos menores que 500000 elementos, porém para conjuntos maiores que isso a entrada decrescente passa a ter um número maior de trocas. Acredita-se que este comportamento ocorre devido ao conjunto com 1 milhão de elementos dispostos aleatoriamente por coincidência não seja necessário tantas trocas para ordena-lo.

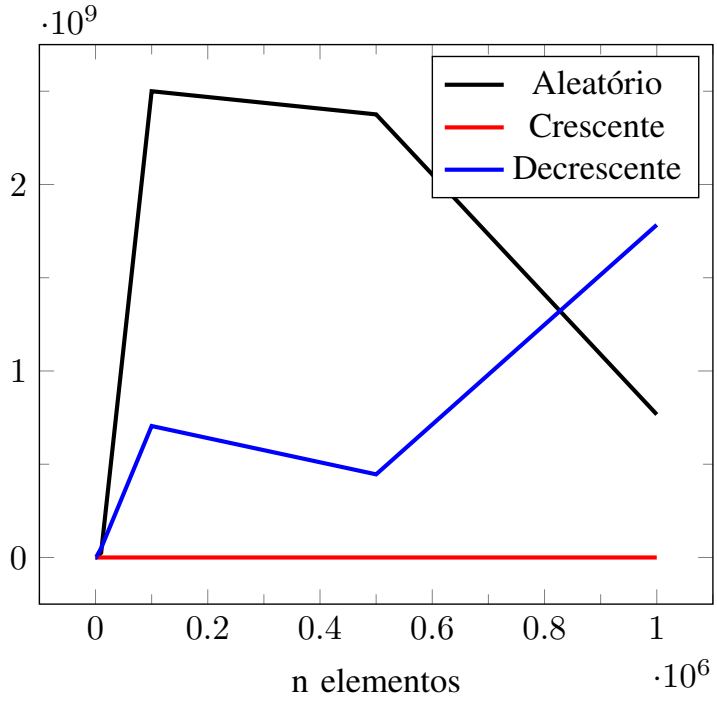

Fig. 28. Número de trocas realizadas pelo algoritmo Bubble Sort melhorado.

2) Quick Sort Pivô inicial: O algoritmo Quick Sort com implementação colocando sempre o pivô como ponto inicial tem números de trocas que apresentam comportamento linear. O maior número de trocas ocorre quando a entrada está organizada em ordem aleatória, entretanto os casos em que a entrada está em ordem crescente ou decrescente a quantidade de trocas é a mesma para essas duas entradas, vale ressaltar ainda que a quantidade de trocas na entrada em ordem aleatória é bem maior que a quantidade de trocas nas duas outras configurações de entrada. Observe o Gráfico 29

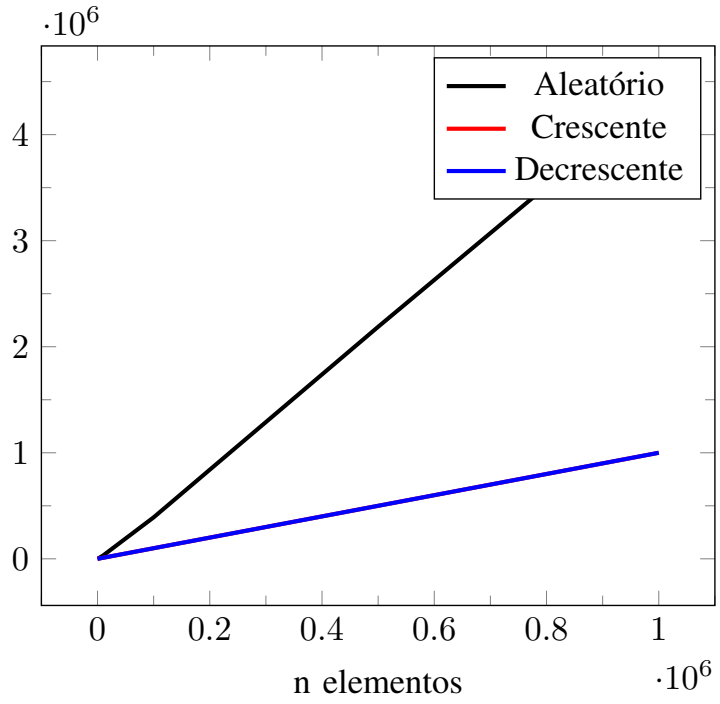

Fig. 29. Número de trocas realizadas pelo algoritmo Quick Sort implementação com pivô inicial.

3) Quick Sort Pivô Central: Quando o algoritmo Quick Sort é implementado com o pivo sendo sempre o elemento central o comportamento de número de trocas é diferente a implementação com pivô sendo sempre o inicial. Neste caso, o maior número de trocas absoluto continua sendo para a entrada organizada de modo aleatório, contudo a diferença entra a implementação discutida anteriormente está que o número de trocas para entradas em ordem crescente e decrescente é diferente, tendo maior quantidade de trocas para a decrescente e menor para a entrada em ordem crescente. Observe o Gráfico 30.

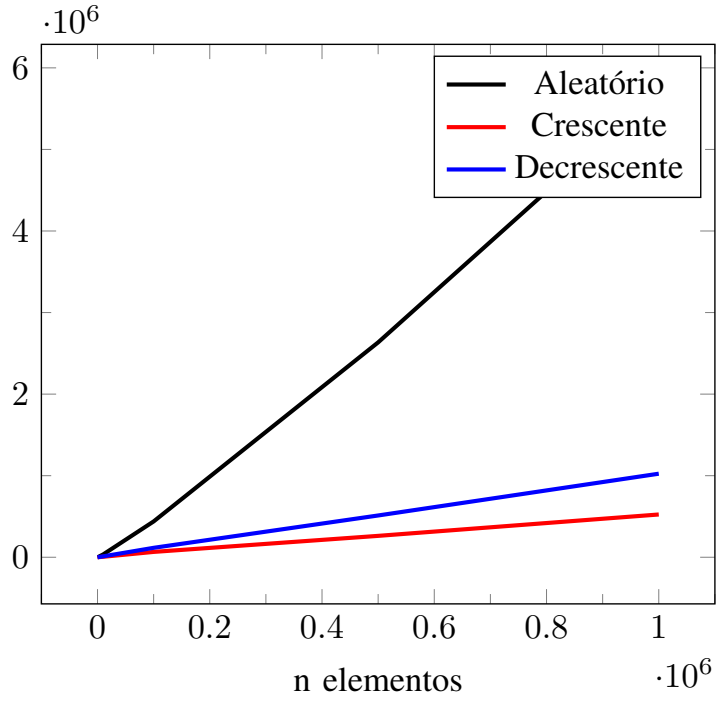

Fig. 30. Número de trocas realizadas pelo algoritmo Quick Sort implementação com pivô central.

4) Shell Sort: O Shell Sort não realiza nenhuma troca quando a entrada de dados está em ordem decrescente, a verificação presente no corpo do algoritmo garante isso. Contudo, para os outros demais casos existem uma quantidade de trocas significativa de comportamento não linear, porem quanto maior a dimensão da entra maior o número de trocas. A maior quantidade de trocas ocorre com a entrada em ordem decrescente e a segunda maior em ordem aleatória. Observe o Gráfico 31.

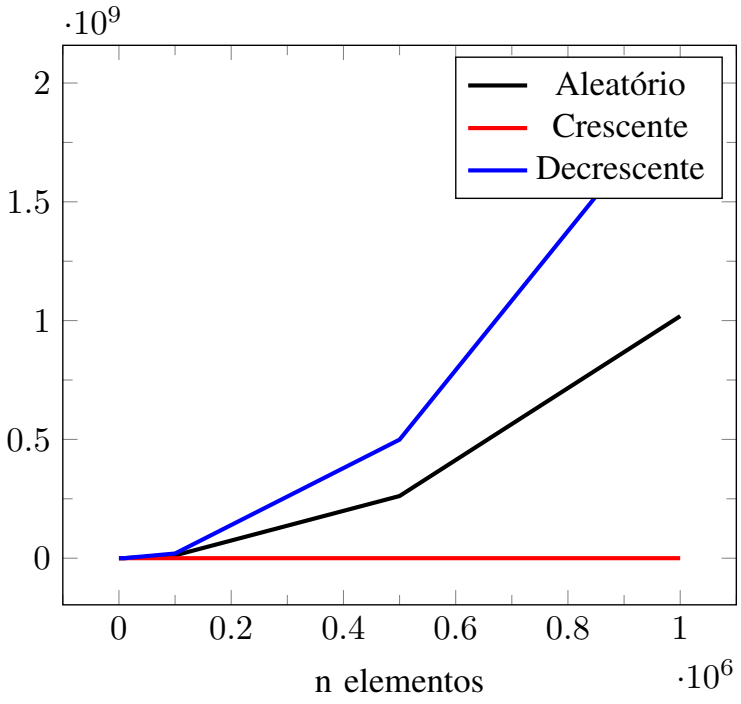

Fig. 31. Número de trocas realizadas pelo algoritmo Shell Sort.

5) Selection Sort: No algoritmo Selection Sort todas as quantidades de trocas tem valores bem espaçados entre as três configurações de entrada. Neste caso, a que menos realiza trocas são quando a entrada está em ordem crescente, neste caso, nenhuma troca é realizada, pois, o algoritmo verifica que os elementos já estão em ordem correta. Já nas duas demais configurações de entrada ocorrem um número de trocas acentuado, sendo o maior deles na entrada em ordem aleatória e a segundo maior na entrada em ordem decrescente. Observe o Gráfico 32.

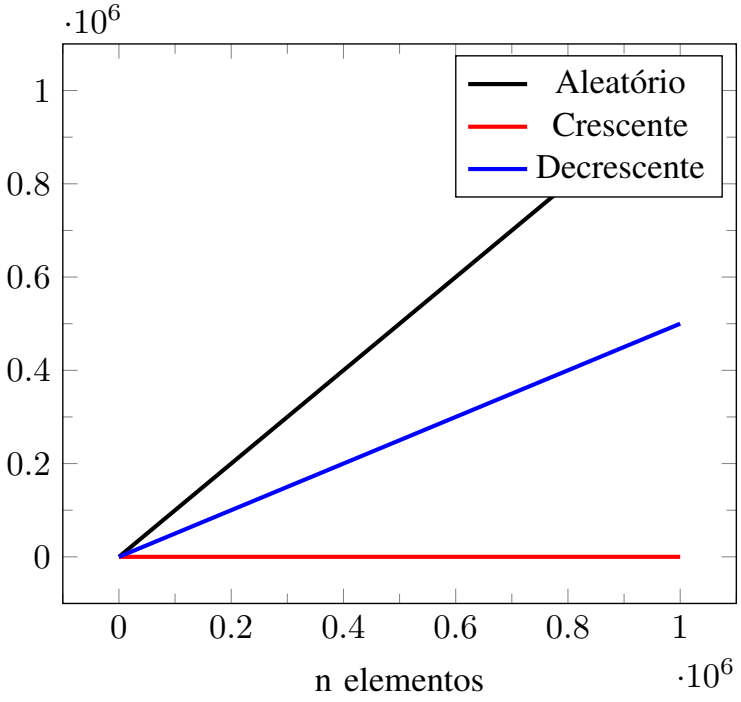

Fig. 32. Número de trocas realizadas pelo algoritmo Selection Sort.

6) Merge Sort: O algoritmo Merge Sort independentemente da configuração de entrada sempre irá realizar o mesmo número de trocas em seu processo de ordenação. Assim, independente se a entrada contem elementos organizados em ordem aleatória, crescente ou decrescente, o número de troca será o mesmo. Vale ressaltar ainda que conforme a quantidade de elementos aumenta, maior será o número de trocas formando um comportamento linear. Observe o Gráfico 33.

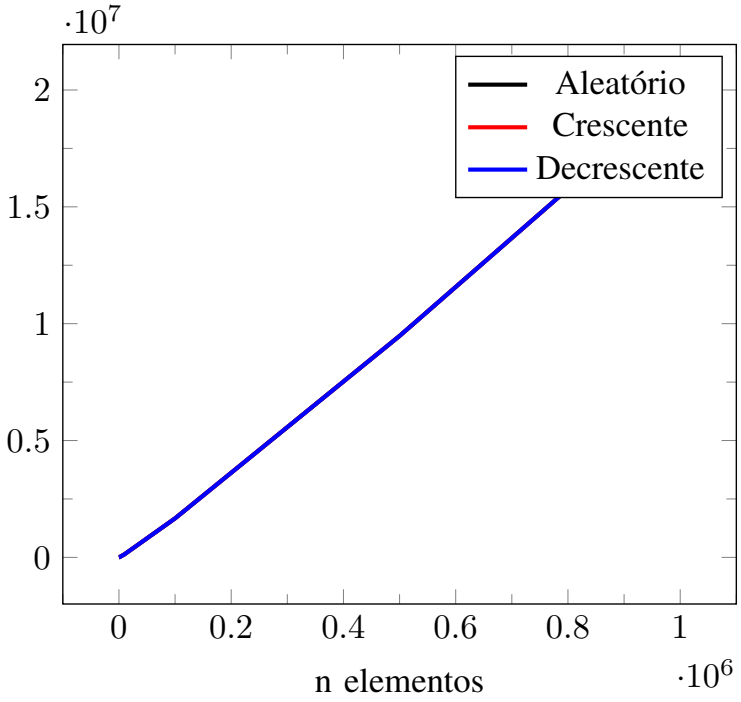

Fig. 33. Número de trocas realizadas pelo algoritmo Merge Sort.

7) Heap Sort: No caso do algoritmo Heap Sort, é constatado também que conforme o número de elementos a serem ordenados aumentam maior será o número de trocas em todas as configurações de entrada, formando um comportamento bem próximo ao linear.

Contudo, existe uma mínima diferença entre a quantidade de trocas em cada configuração de entrada. A menor quantidade de trocas acontece quando a entrada de dados são elementos em ordem decrescente, já a maior quantidade de trocas ocorre quando os elementos está dispostos em ordem aleatória, caso os elementos estejam em ordem aleatória a quantidade de trocas é menor que a crescente e maior que a de ordem decrescente. Observe o Gráfico 34.

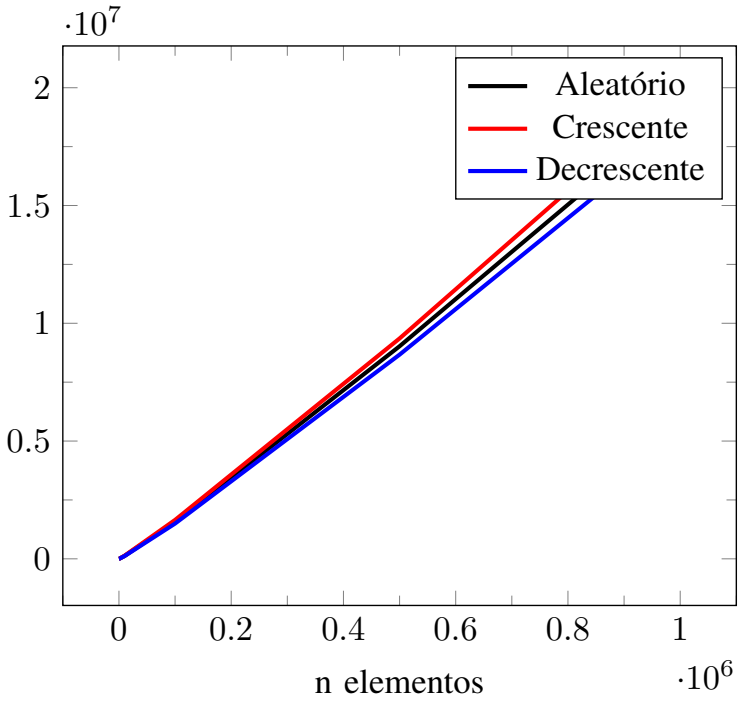

Fig. 34. Número de trocas realizadas pelo algoritmo Heap Sort.

V. Considerações Finais

Em suma, este estudo experimental buscou analisar e comparar o desempenho de diferentes abordagens e algoritmos de ordenação com o intuito de identificar características e melhores situações para utilização de cada método.

Os algoritmos *Quick Sort* com pivô central, *Shell Sort* e *Heap Sort* apresentaram os melhores desempenhos dentro de suas respectivas abordagens de ordenação. Apesar de serem boas opções, vale ressaltar uma análise quanto ao contexto e necessidades da aplicação que tais algoritmos serão empregados, desse modo, desde implementações mais simples como o *Insertion Sort* podem ser empregadas, principalmente em cenários que o conjunto de dados cresce por demanda, que nesse caso, é melhor opção devido a simplicidade e desempenho. Adicionalmente, vale lembrar o algoritmo *Merge Sort* devido a estabilidade quanto a entrada de dados, porém, como ponto negativo, o consumo adicional de memória.

REFERÊNCIAS

[1] ZIVIANI, Nivio. Projeto de Algoritmos: Com implementações em Pascal e C. São Paulo. Cengage Learning, 2011.
[2] BIGGAR, Paul. GREGG, David. Sorting in the Presence of Branch Prediction and Caches Fast Sorting on Modern Computers. University of Dublin, 2005.
[3] SEDGEWICK, Robert. WAYNE, Kevin. Algorithms. 4 Ed. Addison-Wesley. 2001