Aplicativo para análise comparativa do comportamento de algoritmos de ordenação

João Paulo Folador[1]
Lázaro Nogueira Pena Neto[1]
David Calhau Jorge[2]

Resumo: Este artigo apresenta uma ferramenta prática de auxílio pedagógico para demonstrar o comportamento de alguns algoritmos de ordenação. Nesse contexto, foram estudados e confrontados os algoritmos: *bubble sort*, *merge sort*, *quick sort* e o *shell sort*, de modo a ordenar uma matriz de elementos variáveis. Na aplicação desenvolvida é permitido ao usuário configurar a quantidade de elementos da matriz a serem ordenados, quais métodos serão utilizados, a quantidade de ciclos somados e, o resultado final, é mostrado por meio de um gráfico, comparando os métodos aplicados. Ainda, como saída do processo de ordenação, são mostrados: o tempo médio de execução, o desvio padrão, o tempo mínimo e o tempo máximo gasto para ordenar os elementos da matriz. Desse modo, essa aplicação de apoio didático torna-se prática para testes feitos pelos alunos nas disciplinas que envolvem o estudo de algoritmos de ordenação facilitando, assim, o processo de aprendizagem.

Palavras-chave: Complexidade de algoritmos. Métodos de ordenação. Otimização.

Abstract: *This paper presents a practical tool for teaching aid to demonstrate the behavior of some sorting algorithms. Thus, we studied and compared the algorithms: bubble sort, merge sort, quick sort and shell sort, to sort an array of variable elements. In the developed application the user is allowed to configure the number of array elements to be sorted, what methods will be used, the amount of added cycles and the final result is shown by a graph comparing the methods applied. Even as output of the ordering process are shown: the average execution time, the standard deviation, the minimum time and maximum time taken to sort the array elements. Thus, this application of didactic support becomes practice for tests by students in disciplines involving the study of sorting algorithms, thus facilitating the learning process.*

Keywords: *Algorithm complexity. Ordination methods. Optimization.*

1 Introdução

Sabe-se que os diversos tipos de métodos de ordenação buscam aprimorar o desempenho nos variados tipos de buscas e/ou solucionar um novo tipo de problema de ordenação. Nesse contexto, iniciar a ideia de ordenação nos estudantes é uma boa tarefa. Pensando em facilitar essa tarefa, este trabalho visa à implementação de um software para realizar a comparação entre alguns algoritmos de ordenação, com o intuito de verificar o desempenho em tempo gasto nessas ordenações de matrizes e mostrar o resultado de forma fácil e rápida.

Focado nessa ideia, utilizou-se a IDE C++ Builder para criar um software desktop que atendesse a esses anseios. Para tal, utilizou-se uma matriz com tamanho regulável com o intuito de verificar, também, a resposta de cada algoritmo de ordenação frente a variação da quantidade de elementos presentes nessa matriz. E, com isso, poder comparar os resultados obtidos com o que há na literatura.

[1] Programa de Mestrado Profissional em Inovação Tecnológica, UFTM, ICTE –Universidade – Uberaba - MG – Brasil. {jpfolador@gmail.com; lazaro_nogueira@outlook.com}
[2] Universidade Federal do Triângulo Mineiro, Depto. de Engenharia Elétrica– Universidade — Uberaba - MG – Brasil. {david@eletrica.uftm.edu.br}

http://dx.doi.org/10.5335/rbca.2014.3792

Na literatura existem vários tipos de métodos focados na ordenação de matrizes que contenham elementos numéricos ou alfanuméricos. Neste trabalho, os algoritmos utilizados para realizar a ordenação foram: o bubble sort, merge sort, o quick sort e o shell sort, os quais serão melhor explicados no próximo tópico. A aplicação desenvolvida com esses métodos permite ao estudante visualizar a comparação entre eles, variar a quantidade de elementos presentes na ordenação e/ou praticar testes em computadores com hardwares diferenciados.

O principal intuito da aplicação é permitir ao estudante uma aprendizagem, usando estratégia ascendente que possibilita ao estudante fazer associações ou mapas mentais, relacionando com as situações vividas, trabalhando o raciocínio gradualmente [5]. Desse modo, ao receber a real importância de se ter um recurso visual da aplicação em relação ao comportamento dos algoritmos de ordenação estudados, o aluno passa a compreender melhor as características dos métodos.

O artigo está dividido da seguinte forma: na seção 2 serão mostrados os fundamentos dos métodos usados no software desenvolvido; já na seção 3 tem-se a metodologia subdivida em uma explicação prévia da implementação e posteriormente os testes feitos, demostrando o uso e as características da aplicação; a seção 4 contém os resultados e as discussões; e, por fim, na seção 5, é descrito as principais conclusões.

2 Fundamentos dos métodos de ordenação

2.1 Bubble sort

O bubble sort é um dos algoritmos mais simples de ordenação, o qual, em linhas gerais, funciona passando repetidamente por meio dos elementos a ser classificados, segue comparando cada par dos elementos adjacentes e realiza a troca desses elementos caso estejam na ordem errada. A verificação dos elementos a ser ordenados continua repetidamente até que não precise fazer mais nenhuma troca. Isso indica que os elementos estão então ordenados [2].

Esse algoritmo recebe tal nome, pois a cada passagem ele faz "flutuar" para o topo o maior elemento da sequência. Veja a seguir um exemplo simples, envolvendo uma sequência numérica. Considere uma matriz unidimensional de cinco elementos números igual a: (10 2 8 4 16), ao executarmos a ordenação, utilizando o método bubble sort, teríamos, de forma resumida, os seguintes passos:

primeiro passo:
(**10 2** 8 4 16) → (**2 10** 8 4 16), compara os dois primeiros se 10 > 2 e faz a troca;
(2 **10 8** 4 16) → (2 **8 10** 4 16), novamente verifica que 10 > 8 e realiza a troca;
(2 8 **10 4** 16) → (2 8 **4 10** 16), nesta etapa também há a troca de 10 por 4, pois 10 > 4;
(2 8 4 **10 16**) → (2 8 4 **10 16**), neste caso 10 < 16, portanto, não há troca dos elementos.

segundo passo:
(**2 8** 4 10 16) → (**2 8** 4 10 16), segue comparando os elementos e caso seja maior há a troca;
(2 **8 4** 10 16) → (2 **4 8** 10 16), aqui, 8 > 4, portanto é necessário fazer a troca dos elementos;
(2 4 **8 10** 16) → (2 4 **8 10** 16), sem troca;
(2 4 8 **10 16**) → (2 4 8 **10 16**) sem troca também.

Agora, verifica-se que a lista de elementos está ordenada. Contudo, o algoritmo não sabe se está tudo ordenado, ele precisa passar pela lista novamente e não realizar nenhuma troca para, aí sim, saber que está tudo certo.

terceiro passo:
(**2 4** 8 10 16) → (**2 4** 8 10 16)
(2 **4 8** 10 16) → (2 **4 8** 10 16)
(2 4 **8 10** 16) → (2 4 **8 10** 16)
(2 4 8 **10 16**) → (2 4 8 **10 16**)

Desse modo, no terceiro passo não houve nenhuma troca e o algoritmo bubble sort é, então finalizado e garante a ordenação da lista de elementos.

Para o desempenho do bubble sort, tem-se: no pior caso e no médio caso $O(n^2)$, ou seja, uma complexidade quadrática, já no melhor caso O(n) a complexidade é linear. Ao se analisar o melhor caso desse algoritmo, percebe-se que, caso a quantidade de elementos a serem ordenados dobrar, o tempo de execução também dobrará. Já no seu pior e médio caso, se o tamanho do matriz dobrar, o tempo de execução passa a ser quatro vezes maior. Desse modo, esse método não obterá bons resultados em volumes grandes de dados [10].

A seguir, é mostrado o pseudocódigo do bubble sort simples, usado para implementação na aplicação deste trabalho.

```
1. Função bubble sort (inteiro mat[tamanho])
2.    tempo = 0
3.    t = inicia contagem do tempo
4.    inteiro aux = 0;
5.    inteiro k = tamanho - 1;
6.    Para i = 0 até tamanho passo 1
7.          k = k - 1
8.          Para j = 0 até k passo 1
9.            Se mat[j] > mat[j+1] então
10.                aux = mat[j];
11.                mat[j] = v mat[j+1];
12.                mat[j+1] = aux;
13.            fim se
14.        fim para
15.    fim para
16.    t = tempo final - t;
17.    retorna tempo;
```

Nesse sentido, obtém-se, por meio dessa função, o tempo gasto para ordenar os elementos de uma matriz de tamanho definido, usando o método bubble sort. A seguir, será tratado o método Merge sort.

2.2 Merge sort

O algoritmo Merge sort é um exemplo de algoritmo de ordenação do tipo dividir e conquistar. Sua ideia básica consiste em dividir (o problema em vários subpartes e resolvê-las por meio da recursividade) e conquistar (após todas essas subpartes terem sido resolvidos ocorre a conquista que é a união das resoluções). Como o algoritmo do merge sort usa a recursividade em alguns problemas, essa técnica não é muito eficiente devido ao alto consumo de memória e tempo de execução [9].

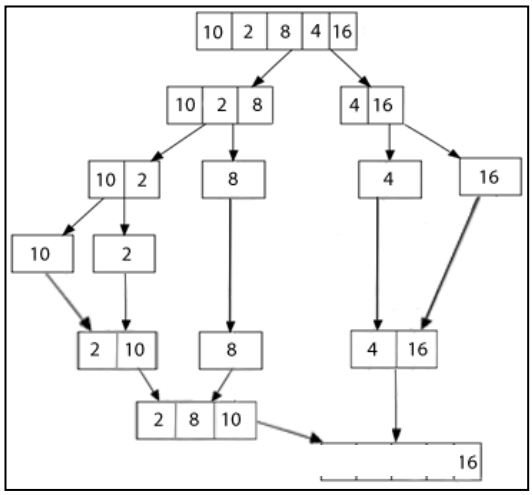
Os três passos úteis dos algoritmos dividir e conquistar, ou *divide and conquer*, que se aplicam ao merge sort são:

1. dividir os dados em subsequências pequenas;
2. classificar as duas metades recursivamente, aplicando o merge sort;
3. juntar as duas metades em um único conjunto já classificado.

Para exemplificar seu funcionamento, considere a matriz unidimensional com cinco elementos utilizado anteriormente: (10 2 8 4 16), na qual será aplicada a ordenação, utilizando o método merge sort, veja Figura 1.

Nesse método, os elementos são subdivididos em sequências menores e posteriormente são unidos em ordem, e tal processo ocorrerá até que toda a matriz esteja em ordem crescente. Para o merge sort comum, sem variações, a complexidade de tempo no pior caso, caso médio e melhor caso é O(n log n) [9]. Assim sendo, caso a matriz dobre de tamanho, o tempo de execução é ligeiramente maior que o dobro. A seguir é representado o pseudocódigo do método merge sort.

Figura 1: Merge sort, adaptado [14]




```
1. Função merge sort (inteiro mat[], inteiro tamanho)
2.   inteiro meio;
3.   Se (tamanho > 1) então
4.          meio = tamanho / 2
5.          merge sort(mat, meio)
6.          merge sort(mat + meio, tamanho – meio)
7.          unir(mat, tamanho)
8.   Fimse
```

Essa função, de forma simples, trata os eventos que compõem as características da ordenação no método merge sort. Posteriormente, será mostrado outro método usado para testar a ordenação de uma matriz.

2.3 Quick sort

Esse algoritmo de ordenação foi criado por volta de 1960 por Charles Antony Richard Hoare. Ele o criou ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras. Focado nesse objetivo, Richard tentava reduzir o problema original em problemas menores, tornando-os mais fáceis e rápidos de serem solucionados [11].

O método de ordenação quick sort utiliza uma estratégia de divisão e de conquista, na qual as chaves são rearranjadas de modo que os menores precedam as chaves maiores [12]. Em seguida, o quick sort ordena as duas partes dos elementos menores e maiores recursivamente até que a lista completa encontre-se ordenada. Para isso, alguns passos são seguidos [13]:

1. escolha um elemento da lista, denominado pivô;
2. rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores que esse, e todos os elementos posteriores ao pivô sejam maiores. Ao fim do processo o pivô estará em sua posição final e haverá duas sublistas não ordenadas. Essa operação é denominada partição;
3. recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores.

Na sequencia, há um exemplo para melhor entendimento do método de ordenação quick sort. Supõe-se uma matriz unidimensional inicial como sendo: (25 57 48 37 12 92), em seguida, são aplicados os passos que caracterizam esse método de ordenação:

(12 25 57 48 37 92), escolhido o pivô 25 a lista é rearranjada nas duas partes;
(12) e (57 48 37 92), os elementos são, então, separados em subproblemas;
(12), essa parte já está ordena já que tem somente um elemento;
(57 48 37 92), o processo é chamado recursivamente para ajustar essa outra parte;

(57 48 37 92), 57 é, então, o próximo pivô;
(48 37 57 92), a lista reordena-se em relação ao novo pivô e posteriormente separados;
(48 37) e (92), neste ponto é, então, feita a chamada recursiva para o primeiro subproblema;
(48 37), escolhe-se o pivô e reordena-se a lista;
(37 48), deste modo todas as partes estão por fim ordenadas;
(12 25 37 48 57 92), após os subproblemas retornaram da recursividade;

Por conseguinte, é mostrado, em pseudocódigo, o formato do algoritmo de ordenação quick sort usado na implementação [1].

```
1.  Função  quick  Sort(inteiro  mat[],  inteiro  esquerda,  inteiro
direita);
2.      inteiro i, j, x, y;
3.      i = esquerda;
4.      j = direita;
5.      x = mat[(esquerda + direita) / 2];
6.      Enquanto (i <= j) faça
7.         Enquanto (mat[i] < x && i < direita) faça
8.            incrementa i
9.        Fim Enquanto
10.        Enquanto (mat[j] > x && j > esquerda) faça
11.             incrementa j
12.        Fim Enquanto
13.        Se (i <= j) então
14.             y = valor[i]
25.             mat[i] = valor[j]
26.             mat[j] = y
27.             incrementa i
28.             incrementa j
29.        Fim Se
30.     Fim Enquanto
31.     Se (j > esquerda) então
32.         quick Sort(mat, esquerda, j)
33.     Fim Se
34.     Se (i < direita) então
35.         quick Sort(mat,  i, direita)
36.     Fim Se
```

No melhor caso, tem-se O(n log n), no caso médio, também, a complexidade é de O (n log n) e no pior caso passa a ter um desempenho de O (n^2). [4] [12]. O próximo e último método implementado neste trabalho foi o shell sort, o qual será discutido no próximo tópico.

2.4 Shell sort

Criada por Donald Shell, em 1959, e baseado em uma sequência incremental, a técnica de ordenação consiste em tratar uma matriz bidimensional ou unidimensional, sendo dividida em *h partes* e ordenada. Isso significa que ao se quebrar em partes existe uma possibilidade de que essas já estejam parcialmente ordenadas. Ou, no pior caso, que todas as partes estejam desordenadas, tendo que percorrer todas as *h divisões*. O interessante é que para a ordenação acontecer, ele atua sobre os elementos mais afastados e depois esse afastamento se reduz, convergindo todos eles até que essa diferença seja mínima, ou seja, todos os elementos já estarão ordenados [8].

De toda forma, esse espaçamento de divisão escolhido que se dá o nome de pivô, deverá ser escolhido de forma que não seja tendencioso. Isso significa que não podemos simplesmente colocar uma sequência predefinida e ordenada para ele. Isso faria com que os dados originais pudessem sofrer alguma distorção que

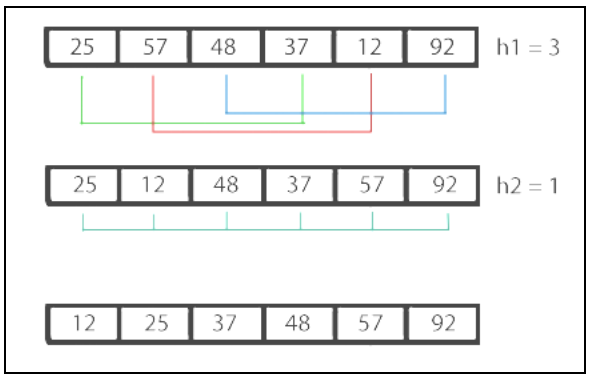
simplesmente favorecesse uma convergência forçada. O elemento chave precisa ser o mais aleatório possível, dentro de uma faixa de tamanho *n* de entradas. [3]

Assim, devemos observar as seguintes características que são variantes na implementação desse algoritmo [6]:

1. o shell sort é baseado em uma variável chamada de incremento de sequência, ou incremento de shell, denominado pela letra h, e durante a execução do algoritmo, é decrementada até o valor 1;
2. utilizando o incremento de shell, o algoritmo passa comparando os elementos distantes em uma matriz, em vez de comparar os adjacentes;
3. a ordenação é realizada em vários passos, usando uma sequência de valores do incremento de shell <h1, h2, h3,… hn> onde se começa por hn seleciona-se apenas os valores que estão hn elementos distantes um do outro, depois ordena-se esses elementos com algum algoritmo de ordenação simples como bolha, seleção ou inserção. Por fim, apenas os elementos selecionados serão ordenados, os outros são todos ignorados.

Para exemplificar o método de ordenação shell sort, considere a matriz unidimensional contendo os elementos (25 57 48 37 12 92), desse modo, ao ordená-los, usando o shell sort, e considerando h1 = 3, h2 = 1, tem-se a ordenação executada de acordo com o que é mostrado na Figura 2.

Figura 2: Exemplo de ordenação usando shell sort



Por conseguinte, foi escrito o método em pseudocódigo da função que utiliza a ordenação shell sort para organizar, de forma crescente de valores, os elementos de uma dada matriz de tamanho variável.

```
1. Função shellsort(inteiro mat[], inteiro tamanho)
2.   inteiro n, j, i, k, m, meio;
3.   n = tamanho;
4.   Para m = n/2 até m > 0 passo m/2 faça
5.       Para j = m até j < n passo 1 faça
6.           Para i = j – m até i >= 0 passo -m faça
7.              Se (mat[i + m] >= mat[i]) então
8.                  sair da função;
9.              senão
10.                  mid = mat[i];
11.                  mat[i] = mat[i + m];
12.                  mat[i + m] = mid;
13.             Fim Se
14.          Fim Para
15.      Fim Para
16.  Fim Para
```

Sua complexidade temporal no melhor caso é de O(n), já no caso médio é variável em relação ao tamanho da sequência, e no pior caso passa a ser de $O(n \log^2 n)$. Esse método de ordenação utiliza uma quebra

sucessiva da sequência a ser ordenada e faz a ordenação por inserção na sequência obtida. Tendo sua complexidade quadrática esse método perde eficiência em sequências muito grandes [6].

2.5 Comparação dos algoritmos

Na Tabela 1 foi feito o resumo da descrição comparativa da notação Big O, o qual descreve o comportamento das funções em relação a algum valor em específico ou quando os valores tendem ao infinito, e permite ao usuário concentrar na taxa de crescimento das funções.

Tabela 1: Resumo da comparação dos métodos usados (adaptado [7])

Método	Caso médio	Pior caso	Estabilidade
Bubble sort	$O(n^2)$	$O(n^2)$	Sim
Merge sort	O(n log n)	O(n log n)	Sim
Quick sort	O(n log n)	$O(n^2)$	Não
Shell sort	variável	$O(n \log_2 n)$	Não

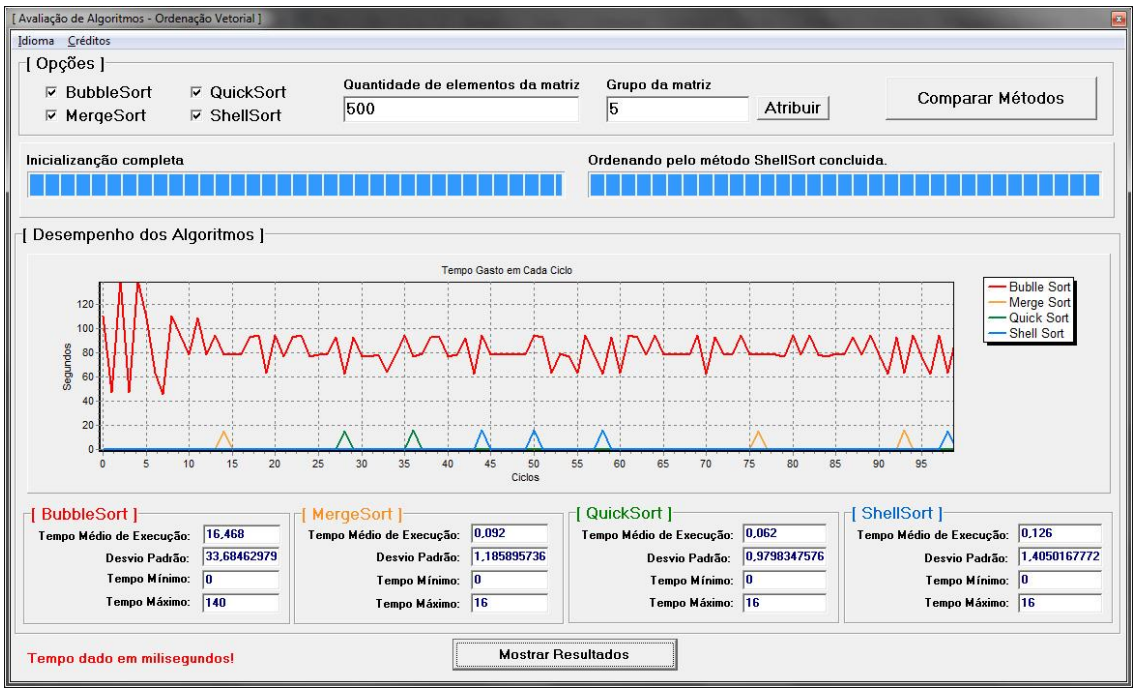
Em linhas gerais, pode-se concluir que no pior caso dos métodos o bubble sort e quick sort terão um comportamento pouco eficiente já que com quantidades grandes de dados o tempo de ordenação também será aumentado consideravelmente quando comparado ao Big O dos outros métodos [7].

3 Metodologia

Após implementação das funções descritas anteriormente, criou-se uma interface para visualizar os testes realizados. Para isso, foi implementado uma matriz quadrada de limite variado ou controlável via código para simular essa capacidade da quantidade de elementos a serem ordenados pelas funções.

O programa conta com um gráfico que mostrará a curva do tempo gasto por cada grupo de matrizes ordenadas. Cada método de ordenação ainda será avaliado quanto ao tempo médio de execução, ao desvio padrão, além do tempo mínimo e do tempo máximo gasto nas execuções. Na aplicação, o usuário pode alterar entre os idiomas português e inglês. A interface básica pode ser vista na Figura 3.

Figura 3: Interface do programa



As configurações básicas para preparar o software são as seguintes: o usuário deve selecionar a opção em relação aos métodos que ele deseja utilizar; após essa escolha ele deve optar pelo preenchimento do campo que contém a quantidade de elementos na matriz, que por padrão está configurada para conter 1.000 x 1.000 elementos; posteriormente, o grupo da matriz deve ser informado, visto que essa opção serve para fazer uma média do tempo da quantidade de linhas agrupadas. Clica-se no botão *atribuir* para validar as configurações e, por fim, o botão chamado *comparar métodos* deve ser clicado para dar início ao processo.

Ao iniciar, a aplicação limpa todos os elementos da matriz, e logo em seguida atribui, de forma randomizada, valores aleatórios inteiros para cada campo da matriz. Desse modo, tem-se ao final desse processo, uma matriz quadrada com a quantidade de elementos escolhidos e desordenada. Por conseguinte, os métodos escolhidos são usados para ordenar essa matriz. Vale lembrar que existe uma matriz idêntica para cada método separadamente, ou seja, cada método ordenará a mesma disposição dos elementos inicialmente criados sem interferência de um método no outro. O tempo gasto para ordenar cada linha é, então, guardado para futuros cálculos.

Ao finalizar essa etapa do software, a ordenação, é preciso que o usuário clique no botão denominado *mostrar resultados*, o qual utilizará os tempos de ordenação de cada método para calcular as respectivas informações. E, além disso, plotar o gráfico para cada um dos métodos, proporcionando assim, um retorno ao usuário da aplicação. Após a implementação dessa ferramenta, foi decidido torná-la de código aberto para futuros desenvolvimentos colaborativos, aumentado, assim, a qualidade e a abrangência do projeto. Sendo assim, está disponível para download no link: http://sourceforge.net/p/compararalgoritmosordenacao.

4 Resultados e discussões

Para demonstrar o uso da aplicação e, também, comparar a teoria de cada um dos métodos optou-se por fazer alguns testes de ordenação. Para tal, utilizou-se um microcomputador com processador i7, com 8Gb de memória RAM para executar a aplicação. Não foram trabalhados elementos de estrutura de armazenamento de dados em disco, apenas execução em memória volátil.

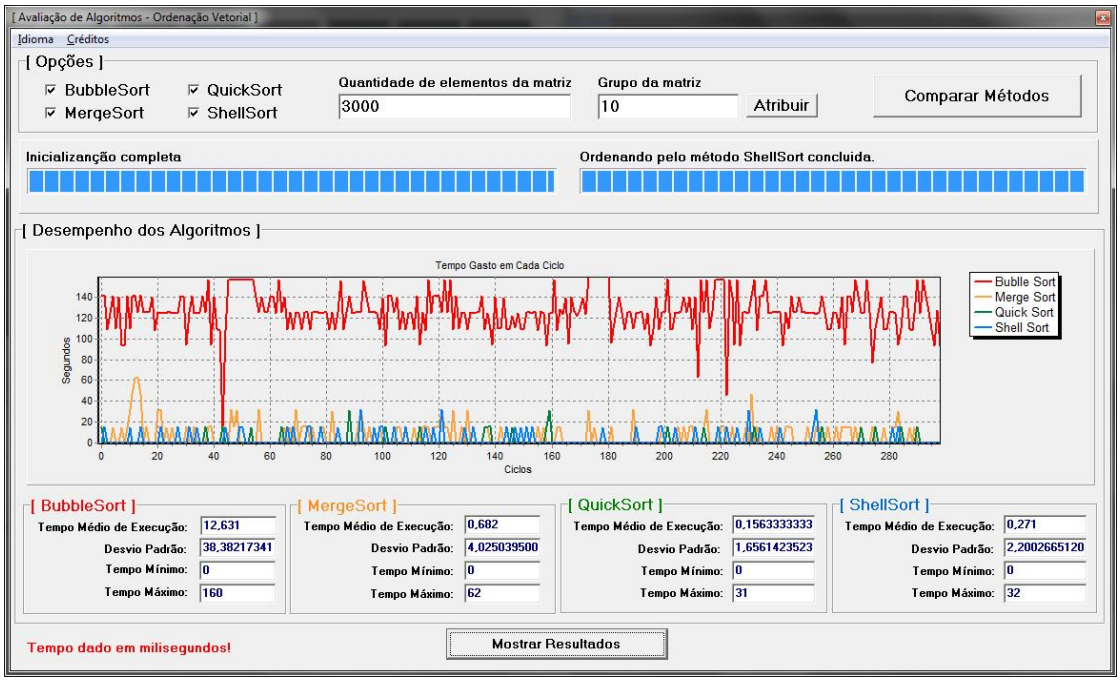
Para a realização do primeiro teste de ordenação a aplicação foi configurada para executar os quatro métodos implementados, a matriz foi configurada com 1.000 x 1.000, o grupo da matriz foi deixado como padrão em dez, ou seja, a cada dez linhas da matriz é feito uma média do tempo gasto e, por fim, foi executado a comparação dos métodos e mostrado os resultados de acordo com a Figura 4.

Figura 4: Teste 1 – todos os métodos ordenando uma matriz quadrada de 1000 linhas e 1000 colunas

Para este primeiro teste, pode-se notar, ainda na Figura 4, que o valor de tempo médio de execução do método bubble sort gasto para ordenar os elementos da matriz, foi o maior em relação aos outros métodos, convergindo para o desempenho mostrado anteriormente. Os demais algoritmos mostraram um desempenho parecido, porém o quick sort apresentou o melhor desempenho para essa quantidade de elementos. O tempo mínimo marca o menor tempo de cada método para ordenar cada linha da matriz. Os valores que apresentaram zero reproduzem o tempo na casa dos microssegundos que não foram possíveis registrar. Já, o tempo máximo é o maior tempo para ordenar uma linha inteira da matriz. O desvio padrão serve para detalhar quanto tempo se dispersou da média, portanto, quanto maior o valor do desvio-padrão mais o tempo desvia-se da média.

O próximo teste foi feito também usando os quatro métodos, porém, aumentou-se o tamanho da matriz quadrada de 1.000 para 3.000, ou seja, uma matriz com 3.000 linhas por 3.000 colunas. Manteve-se o grupo da matriz configurado em 10.

Figura 5: Teste 2 – todos os métodos ordenando uma matriz quadrada de ordem 3.000



Na Figura 5, pode-se conferir a configuração do Teste 2. Após a finalização da ordenação de toda a matriz, percebe-se um aumento normal no tempo médio gasto para ordenação da matriz por todos os métodos. Foi mantido como o método menos eficiente o bubble sort e o mais eficiente o quick sort. Contudo, o shell sort apresentou uma melhor eficiência se comparado ao merge sort para essa nova configuração, como era de se esperar.

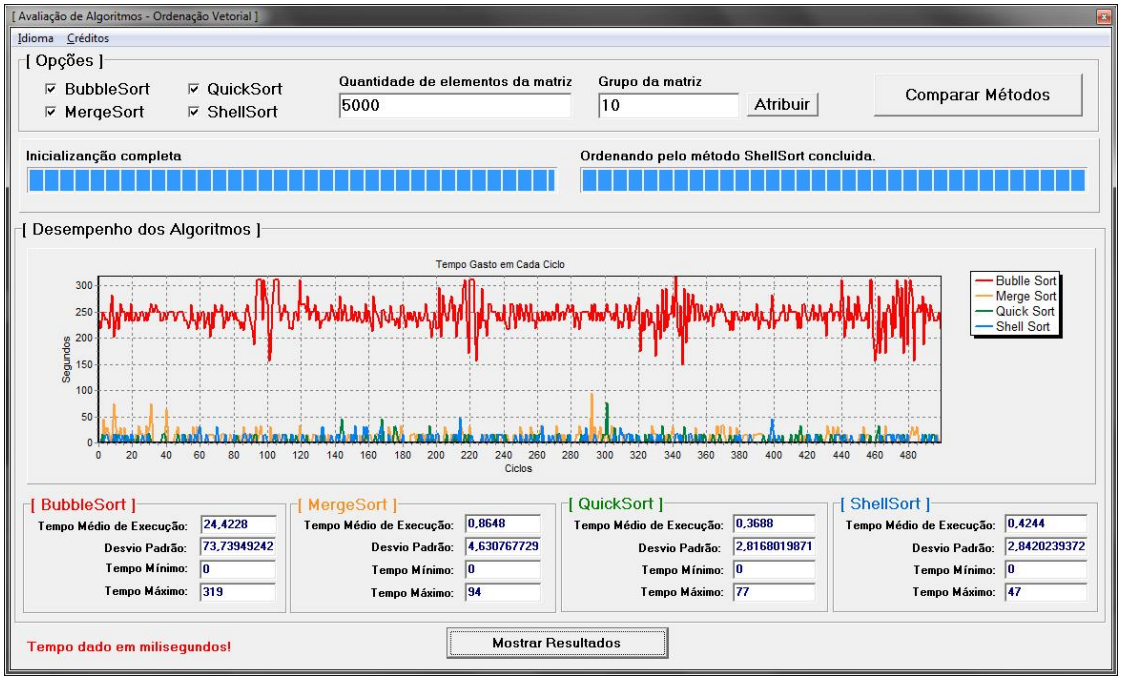
Por fim, optou-se por realizar o Teste 3 com a configuração da matriz em ordem 5.000. Manteve-se a mesma quantidade de agrupamento da matriz, ou seja, em dez linhas e, também, foram utilizados todos os métodos implementados para a comparação dos desempenhos, conforme evidenciam os resultados transcritos na Figura 6.

Nesta etapa, percebe-se que o método de ordenação quick sort não sofreu quase nenhuma alteração em seu tempo de execução, isso mostra o quão eficiente é esse método ao se aumentar, de forma considerável, a quantidade de elementos a serem ordenados. Já o bubble sort, para essa configuração, praticamente dobrou o seu tempo médio de execução para ordenar os elementos, mostrando que à medida que se aumenta a quantidade de elementos ele torna-se mais ineficiente. Os métodos merge sort e shell sort são muito próximos no desempenho, contudo o segundo apresenta melhor desempenho para volumes maiores de dados, mesmo apresentando um componente de não estabilidade.

Outro fato interessante é ressaltado pelo estímulo visual apresentado pela ferramenta, no qual é possível notar, por meio do gráfico, como o método bubble sort, em vermelho, varia seus tempos de ordenação e como

gasta mais tempo na execução das ordenações. Já os demais métodos, praticamente misturam-se, na base do gráfico, por apresentarem um desempenho próximo para as configurações apresentadas.

Figura 6: Teste 3 – todos os métodos ordenando uma matriz quadrada de ordem 5000



5 Conclusão

Durante o estudo realizado, pôde-se perceber a real importância de se ter um recurso visual para melhor entendimento dos métodos de ordenação. Essa prática estimula o aprendizado, facilitando a observação imediata do que acontece em cada um dos métodos. Essa observação pode, então, ocorrer pela análise do gráfico e, também, pela comparação numérica dos tempos de execução, tempo mínimo, máximo e desvio padrão.

Além disso, a aplicação proposta permite uma configuração da quantidade de elementos a serem ordenados, deixando o usuário apto a fazer testes variados de ordenação, melhorando, assim, a assimilação do conteúdo teórico. Com os testes realizados para demonstrar o funcionamento da aplicação, pode-se verificar que o método denominado quick sort obteve o melhor desempenho em todos os testes, comprovando, desse modo, sua boa eficiência para volumes maiores de dados. Já o bubble sort clássico é bastante ineficiente se comparado aos demais métodos.

A ferramenta desenvolvida neste artigo busca facilitar a cognição desses métodos estudados nas disciplinas de computação. Futuramente, pensa-se na implementação de outros algoritmos como: InsertionSort, BozoSort, HeapSort e RadixSort, dentre outros. E, também, permitir outros testes, criando uma base de informações como referência de uso. Tanto para determinados casos de hardware e software. Nesse caso, poder-se-ia considerar o desdobramento em termos de espaço de memória, divisão de ciclos seja para CPU e/ou GPU e demais atributos.

Por fim, é necessário ratificar que essa é uma ferramenta desenvolvida com o intuito de ajudar no aprendizado de algoritmos de técnicas de ordenação, portanto foi criada e disponibilizada, de forma gratuita e de código fonte aberto, para que futuros alunos e/ou interessados, colaborem na implementação dessa ferramenta de apoio à educação. A aplicação está disponível para baixar no site http://sourceforge.net/.

Referências

[1] ALIYU, Ahmed M.; ZIRRA, P. B. A comparative Analysis of Sorting Algorithms on Integer and Character Arrays. *The International Jornal of Engineering and Science* (IJES), v. 2, n.7, p. 25-30. 2013.

[2] ASTRACHAN, Owen. *Bubble sort:* an archaeological algorithmic analysis. Computer Science Department, Duke University. Reno, Nevada, USA. 19-23 (2003). Disponível em: <http://www.cs.duke.edu/~ola/bubble/bubble.pdf>. Acesso em: 15 nov. 2013.

[3] CIURA, Marcin. *Best increments for the average case of shell sort*. Department of Computer Science, Silesian Institute of Technology, Akademicka 16, p. 44-100, mês. 2001., Gliwice, Poland. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.9679&rep=rep1&type=pdf>. Acesso em: 5 fev. 2014.

[4] DROSDEK, Adam. *Data structures and algorithms in C++*. 2. ed. California: Brooks/Cole, 2001.

[5] FALCKEMBACH, Gilse A.; ARAUJO, Fabrício V. *Aprendizagem de algoritmos:* dificuldades na resolução de problemas. 2003. Disponível em: <http://www.fabricioviero.com.br/artigos/a4_siie.pdf>. Acesso em: 10 jan. 2014.

[6] JIANG, Tao; LI, Ming; VITÁNYI, Paul. *Average-case complexity of shell sort*. Lecture Notes in Computer Science 1644 (1999), p. 453–462. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.244&rep=rep1&type=pdf>. Acesso em: 15 jan. 2014.

[7] PHONGSAI, Jariya. *Research paper on sorting algorithms*. 2009. Disponível em: <http://goo.gl/IXNRus>. Acesso em: 10 fev. 2014.

[8] PLAXTON, C. G., SUEL, Torsen. *Lower Bounds for Shellsort*. Departament of Computer Science, University of Texas at Austin, 1992. Disponível em: <http://cis.poly.edu/~suel/papers/shell2.pdf>. Acesso em: 10 fev. 2014.

[9] QIN, Song. *Merge sort algoritm*. Department of Computer Science. Florida: Institute of Technology. Melbourne, Florida. Disponível em: <http://cs.fit.edu/~pkc/classes/writing/hw13/song.pdf>. Acesso em: 6 Fev. 2014.

[10] RAJPUT, Ishwari S.; KUMAR, Bhawnesh; SINGH, Tinku. Performance comparison of sequential quick sort and parallel quick sort algorithms. *International Journal of Computer Applications* [0975-8887] 2012 v. 57, p. 9-14, 2012. Disponível em: <http://research.ijcaonline.org/volume57/number9/pxc3883363.pdf>. Acesso em: 20 nov. 2013.

[11] SEDGEWICK, Robert. *Implementing quicksort programs*. Brow University. Programming Techniques. S. L. Graham, R. L. Rivest Editors. p. 847-857 (1978). Disponível em: <http://www.csie.ntu.edu.tw/~b93076/p847-sedgewick.pdf>. Acesso em: 10 dez. 2013.

[12] SOZA POLLMAN, Héctor Juan. *Probabilistic cost analysis of logic programs*. Ingeniare. Rev. chil. Ing., Arica, v.17, n.2, pp. 195-204. Agosto 2009. Disponível em: <http://www.scielo.cl/scielo.php?script=sci_arttext&pid=S0718-33052009000200008&lng=es&nrm=iso>. Acesso em: 5 fev. 2014.

[13] TOSCANI, Laira V., VELOSO, Paulo, A. S. *Complexidade de algoritmos*: análise, projeto e métodos. 2 ed. Porto Alegre: Editora Sagra Luzzatto, 2002.

[14] Wikipedia contributors, *Merge sort*, The Free Encyclopedia. Disponível em: <http://en.wikipedia.org/w/index.php?title=Merge_sort&oldid=595333497>. Acesso em: 15 dez. 2013.