PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Algoritmos e Estrutura de Dados
Professor Kleber Jacques Ferreira de Souza

Arthur Andrade Gonçalves
Assuerio Batista dos Santos
Lucas Braga Ferreira
Marcos Pablo Souza de Almeida

Análise de Algoritmos de Ordenação

Contagem
2019

1. INTRODUÇÃO

Após o estudo e entendimento dos diversos algoritmos de ordenação, analisamos na prática o comportamento desses algoritmos, sendo eles BubbleSort, SelectionSort, InsertSort, MergeSort e QuickSort. Além destes já apresentados foi desenvolvido um sexto algoritmo, o **DualSort**, que consiste na união dos algoritmos BubbleSort e SelectionSort.

Os algoritmos foram testados e avaliados utilizando o arquivo de dados da Airbnb localizado no seguinte link https://1drv.ms/f/s!Ai1mr_X9-Sz7lLYXlG8rF6RpaWD0nQ. O arquivo contém 128000 registros, onde para a realização dos testes mencionados foram distribuídos em vetores de 2000, 4000, 8000, 16000, 32000, 64000 e 12800, executando com tais quantidades, cinco testes com cada algoritmo, tendo ao final a média de tempo gasto em cada caso (médio, melhor e pior), desconsiderando o maior e o menor tempos.

2. ALGORITMOS DE ORDENAÇÃO

2.1. BubbleSort

Algoritmo de fácil implementação e com uma didática facilmente entendida. Consiste em fazer sucessivas comparações dois a dois com os dados, verificado qual deles é o maior, fazendo assim um "borbulho" do maior valor para o final do vetor, ou seja, o maior da coleção valor após as comparações estará no final. É realizado comparações entre os dados até que nenhuma troca seja mais necessária. Em relação as trocas apresenta melhor caso notação **O(1)** quando o vetor já está ordenado e pior caso **O(n²)** quando é necessário realizar trocas com todos elementos.

2.2. InsertionSort

Algoritmo que se baseia em inserir dados em uma área do vetor já ordenada. Realiza tal procedimento, fazendo comparações a partir do segundo elemento do vetor, verificando qual é o maior. Caso seja menor o valor é inserido na sua posição correta dentro da área ordenada. Caso seja maior é inserido à frente do valor comparado dentro da área. É realizado tal processo até que não haja mais trocas para serem feitas. Apresenta notação **O(1)** em relação às trocas no melhor caso e **O(n²)** para o pior caso.

2.3. SelectionSort

Algoritmo de ordenação baseado em selecionar o menor valor do vetor de dados colocando-o no início do vetor. Faz comparações com todos os elementos e no mínimo uma troca a cada comparação, tendo assim um custo linear, tanto pro melhor e pior caso levando em conta suas trocas sendo **O(n)**.

2.4. MergeSort

Algoritmo de ordenação baseado em três passos: **dividir, conquistar e combinar**. A ideia por trás desses passos é dividir o vetor de dados ao meio tendo assim dois outros vetores, realizando esse processo de divisão até que reste apenas um elemento no vetor. Ao chegar neste ponto é comparado este vetor com a outra metade, fazendo todo o processo recursivamente até retornar ao vetor completo com os elementos já ordenados. Explicado melhor os três passos temos:

Dividir: Dividi o vetor em outros dois vetores utilizando a fórmula ((final + inicio) /2). É feito esse processo até que reste apenas um elemento no vetor.

Conquistar: Passo que corresponde à volta recursiva do algoritmo, ordenando os dados.

Combinar: Neste passo é realizado a combinação dos subvetores já ordenados até retornar ao vetor com todos os dados já ordenados

O custo deste algoritmo tanto para trocas e comparações, nos melhor e pior casos é **O(n Log n)**.

2.5. QuickSort

Assim como o MergeSort, o QuickSort é um algoritmo recursivo. Também consiste em "dividir para conquistar". Se difere do Merge pois apresenta a escolha de um valor para ser o pivô da coleção, podendo ser o elemento inicial, do meio ou elemento final. O pivô é selecionado para que ele seja o elemento de comparações entre os outros elementos, onde os valores maiores que o pivô ficam à sua direita os valores menores à sua esquerda. Ao finalizar esse primeiro passo, o vetor é quebrado em outros dois vetores com os valores que ficaram a direita e esquerda do pivô, onde é realizado o mesmo processo de comparações com o pivô desse subvetor. É realizado o processo até que reste apenas um elemento no vetor, fazendo a volta recursiva combinando os elementos. Seu custo é **O(n Log n)** para o melhor caso e caso médio, onde por sua vez no pior caso seu custo já é **$O(n^2)$.**

2.6. DualSort

Este algoritmo foi desenvolvido combinando os algoritmos BubbleSort e SelectionSort. Consiste em "borbulhar" o maior valor para o final do vetor, assim como o BubbleSort e a medida que vai comparando os valores levando o maior valor para o final, é armazenada a

posição do menor valor da coleção, onde ao final é trocado o valor da primeira posição com esse valor. Ao final já terá ordenado tanto o final do vetor quanto o início.

3. TESTES REALIZADOS E ANÁLISE

Para realização dos testes dos algoritmos foi utilizada a máquina com as seguintes especificações:

Fabricante/Modelo:

- Dell 7010MT

Processador:

- Intel Core i5 - 3470
- frequência: 3.20ghz - 3.60ghz
- núcleos/threads: 4/4

Memória Ram:

- tecnologia: DDR3
- capacidade: 2x4(8gb) dual channel
- frequência : 1600mhz

Sistema operacional

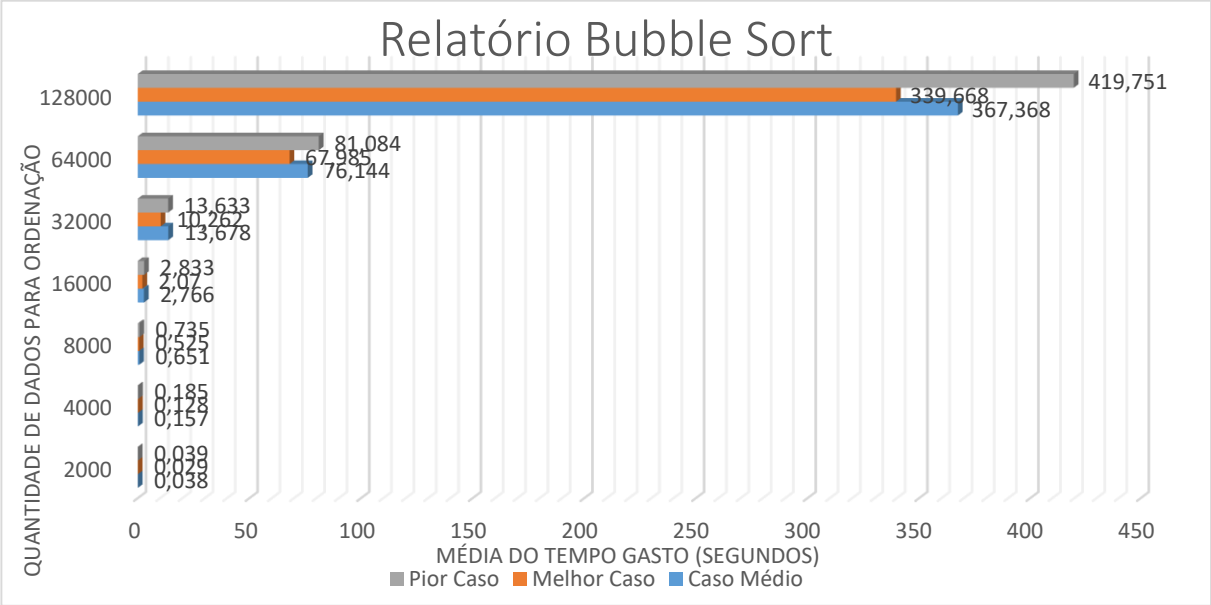
- Win10

Armazenamento

- HD: 1TB

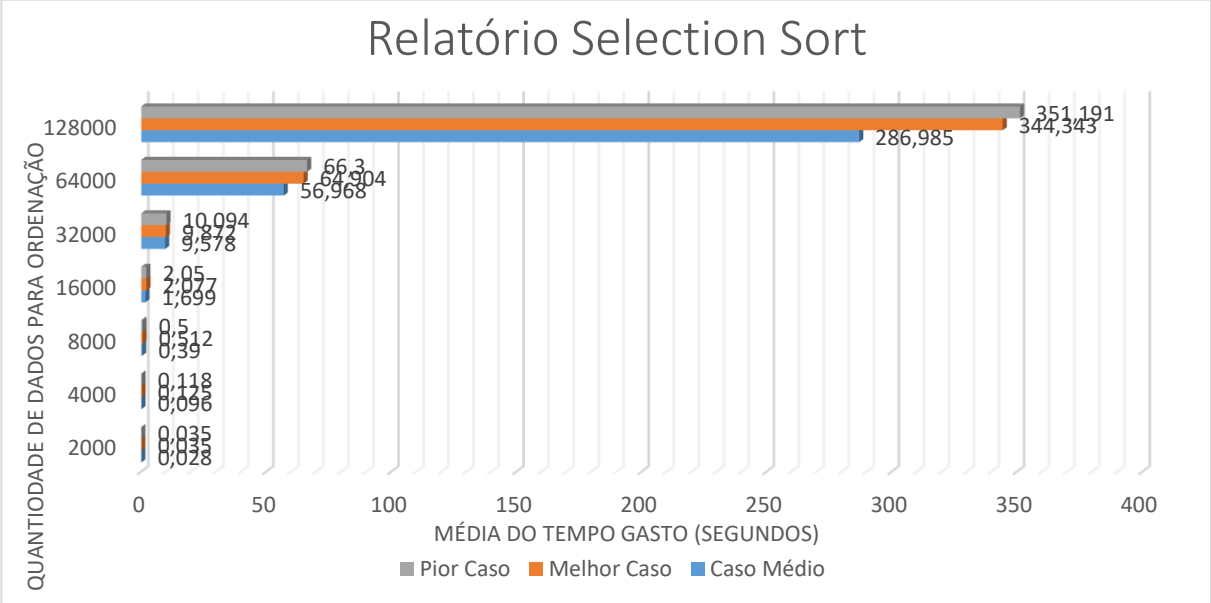
Os gráficos a seguir mostram os resultados dos testes realizados com os algoritmos de ordenação, onde o resultado é a média do tempo ao final de cinco execuções do algoritmo, retirando o menor e maior valor encontrados.

- BubbleSort



Nota-se que em todos os cenários analisados, o caso médio e o pior caso foram semelhantes, sendo ambos compreendidos por *O(N²).* No pior caso, todos os dados foram comparados e trocados de posição, pois estavam na ordem contrária do desejado. No caso médio, ocorreram aproximadamente *O(N²/2)* comparações e trocas. Já no melhor caso, os dados foram apenas comparados, não sendo alterados de posição no vetor, pois estavam previamente ordenados.

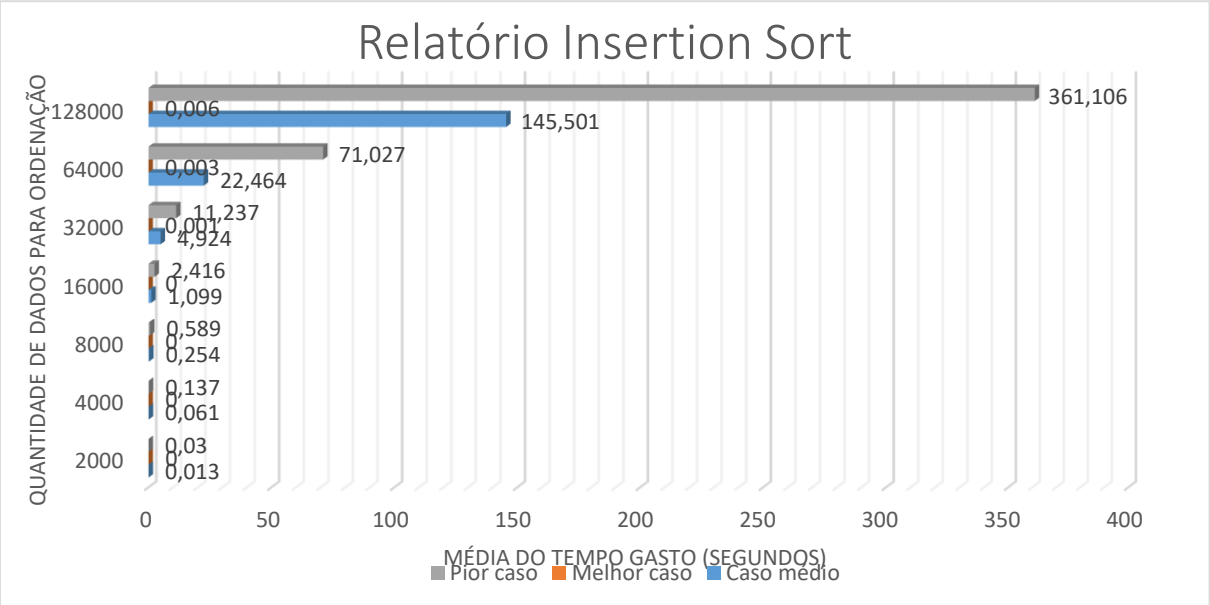
- SelectionSort



Por se tratar de bom algoritmo para lidar com pequenas quantidades de dados, é possível notar uma grande diferença de tempo ao se comparar distintos tamanhos de vetor analisados.

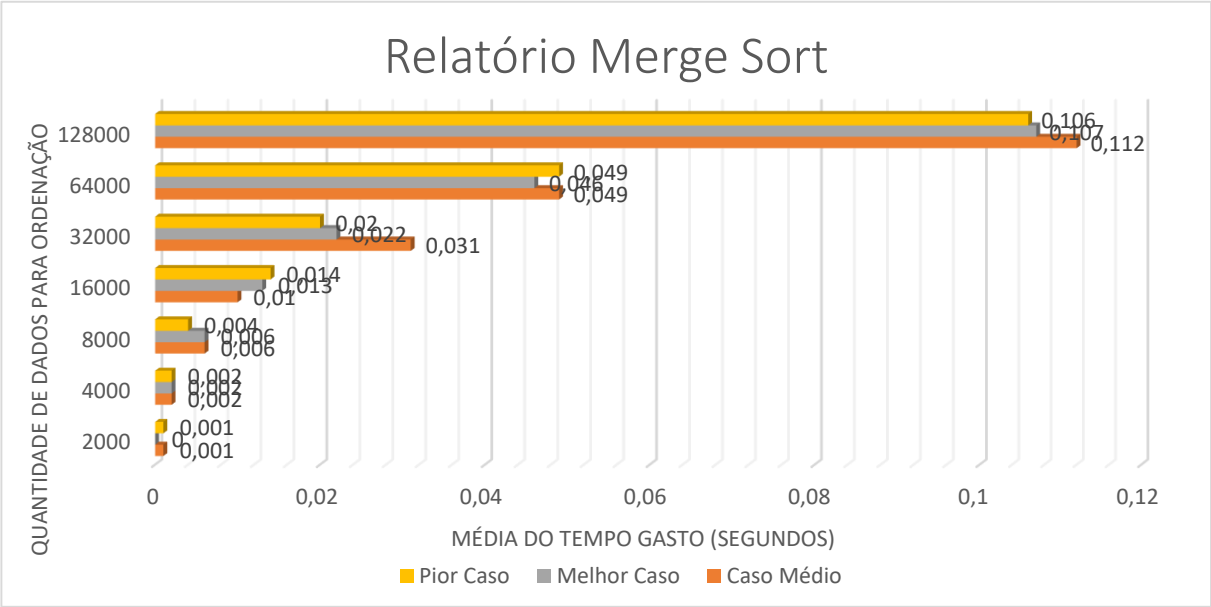
Ademais, tendo em vista que o nível de complexidade é o mesmo para os três casos, sendo esse ***O(n²),*** o tempo gasto foi consideravelmente similar, tanto no pior, melhor e no caso médio. É válido também ressaltar que esse comportamento foi ocasionado devido ao fato de que mesmo o que o vetor já esteja ordenado, o algoritmo fará $(n^2 - n)/2$ comparações, a fim de validar a correta ordenação dos dados.

- **InsertionSort**



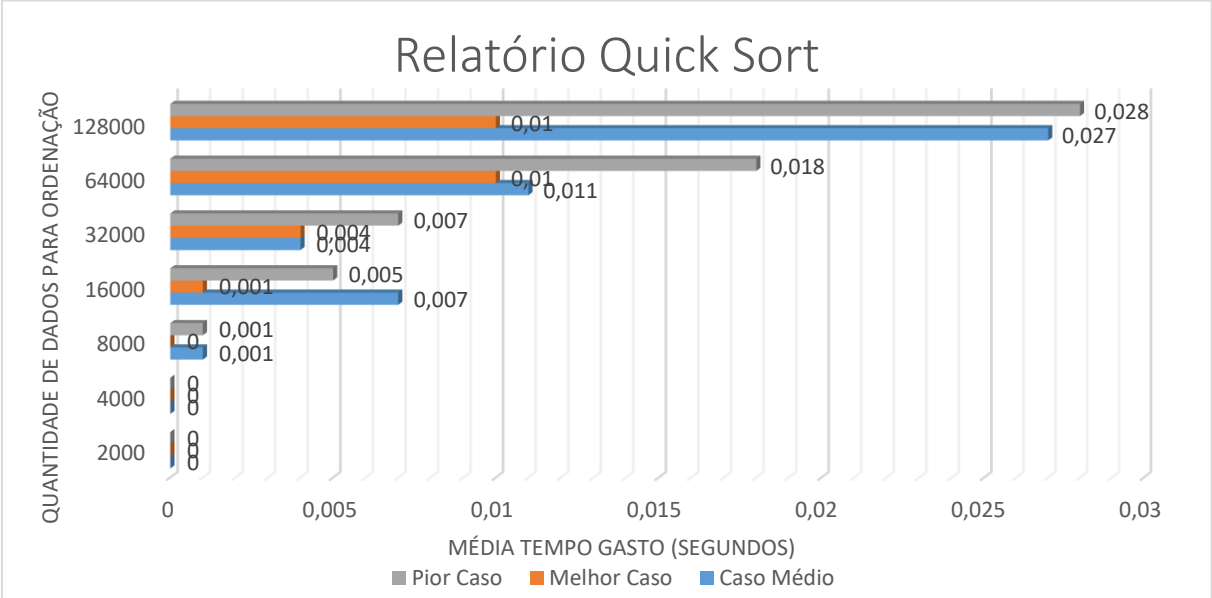
Os resultados apresentados no gráfico condizem com o nível de complexidade do algoritmo, sendo esses de ***O(n)*** para o melhor caso, ***O(n²)*** para o pior caso e de ***O(n²/2)*** para o caso médio. O algoritmo efetua um elevado número de movimentações e trocas em seu pior caso, pois todos os seus elementos precisam ser alterados de posição. Ademais, apresentou resultados superiores se comparado aos algoritmos de ordenação Bubble Sort e Selection Sort.

- **MergeSort**



Em suma, apresentou ótimos resultados no que se refere ao tempo de execução. Tal comportamento pode ser justificado pelo custo de apenas ***O(n log n)*** em todos os casos(pior, melhor e médio). Todavia, é válido ressaltar que pelo fato de criar clones do vetor principal durante sua execução, o consumo de memória foi elevado em relação aos outros algoritmos utilizados.

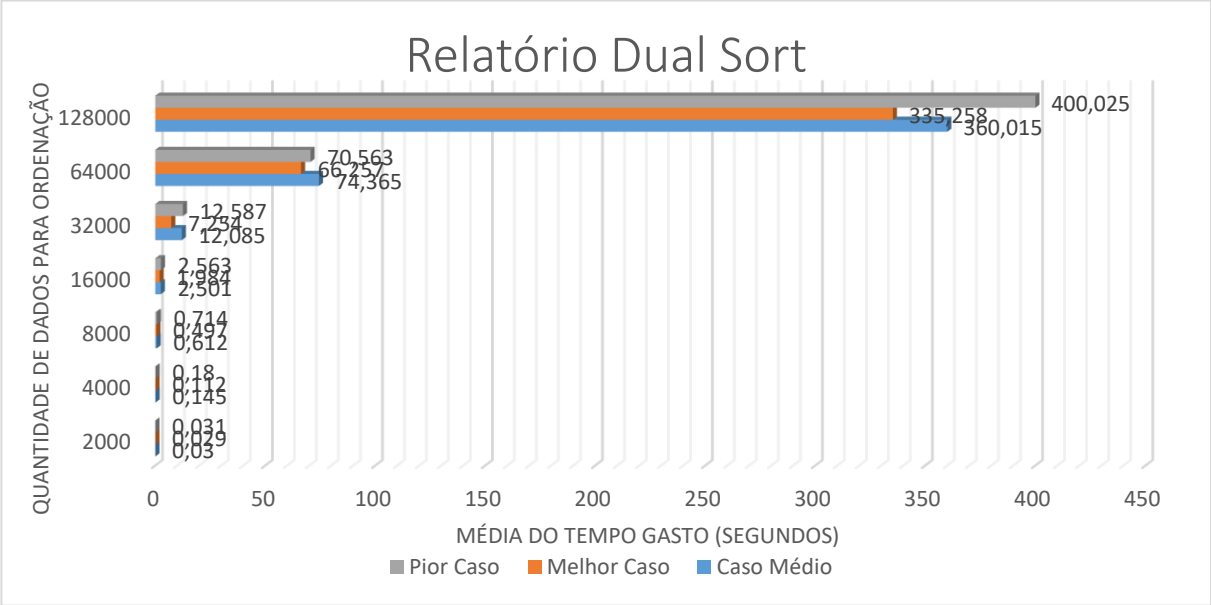
- **QuickSort**



Levando em consideração os níveis de complexidade na casa de O(n log n) para o médio e melhor caso, é possível compreender a superioridade do algoritmo quando comparado aos de ordem quadrática, como o Bubble Sort por exemplo, em que o crescimento é exponencial, já para o QuickSort, o crescimento é logarítmico. No que se refere ao pior caso, acreditamos que

tenha ocorrido algum tipo de otimização feita pelo compilador, tendo em vista que mesmo alterando a posição do pivô para o teórico pior caso, os resultados foram extremamente parelhos ao caso médio, o qual é compreendido por $O(n^2)$, e deveria ter gerado resultados próximos aos apresentados no Bubble Sort.

- **DualSort**



Tendo como base o comportamento quadrático do algoritmo em todos os seus casos, os resultados apresentados assemelharam-se ao que foi entregue no Bubble Sort. Os dados são ordenados mutuamente, em que os maiores valores são borbulhados para o final do vetor, e após isso, o menor valor encontrado é realocado para o início do vetor, fazendo comparações sucessivas até se encontrarem no elemento do meio.

4. CONCLUSÃO

Dentre as condições de testes, o algoritmo de ordenação QuickSort obteve o melhor desempenho. O algoritmo MergeSort é tão rápido quanto, porém utiliza maior quantidade de memória da máquina, por fazer cópias dos dados.

Levando em conta o melhor caso de ordenação, o algoritmo InsertSort obteve melhor desempenho, porém seu pior caso equivale em complexidade ao algoritmo BubbleSort, não sendo assim o melhor algoritmo testado.

O algoritmo desenvolvido pelo grupo DualSort, embora faça trocas simultâneas no início e no fim do vetor, apresenta desempenho equivalente ao BubbleSort, porém com uma leve otimização.

Portanto, reiteramos que dentre os algoritmos de ordenação interna analisados neste documento, o Quick Sort apresentou os resultados mais satisfatórios, curiosamente mantendo a

característica complexidade de O(n log n) em seu pior caso, que entretanto, deveria ter sido semelhante ao resultado obtido no Bubble Sort, na casa de $O(n^2)$. Todavia, acreditamos que tal comportamento inesperado foi o provocado por uma possível otimização automática do compilador, o qual pode ter embaralhado os dados, forçando-os então a ter o mesmo custo que o caso médio.

REFERÊNCIAS

Todo material apresentado nas aulas de Algoritmos e Estruturas de Dados.

STACK EXCHANGE. **QuickSort** . Disponível em: <https://codereview.stackexchange.com/questions/142808/quick-sort-algorithm >. Acesso em: 28 mar.2019.

STACK EXCHANGE. **Exemplo de método recursivo QuickSort em C#**. Disponível em: <https://code.msdn.microsoft.com/windowsdesktop/Exemplo-de-mtodo-recursivo-1f51a7d8>. Acesso em: 20 mar.2019.

STACK EXCHANGE. **Exemplo de método recursivo QuickSort em C#**. Disponível em: <http://dainf.ct.utfpr.edu.br/~maurofonseca/lib/exe/fetch.php?media=cursos:if63c:if63ced_08_ordenacao.pdf>. Acesso em: 16 mar.2019.