UNIVERSIDADE FEDERAL DO MARANHÃO – UFMA

CIÊNCIA DA COMPUTAÇÃO

EMILY JULIANA COSTA E SILVA

Atividade Prática 1

**Algoritmos de Ordenação para serem utilizados na etapa de ordenação das arestas realizada pelo algoritmo de Kruskal.**

SÃO LUÍS - MA

2019

UNIVERSIDADE FEDERAL DO MARANHÃO – UFMA

EMILY JULIANA COSTA E SILVA

Atividade Prática 1

**Algoritmos de Ordenação para serem utilizados na etapa de ordenação das arestas realizada pelo algoritmo de Kruskal.**

Trabalho avaliativo entregue para obtenção de 30% da nota referente à 1ª Avaliação da disciplina de Estrutura de Dados II, do curso de Ciência da Computação.

SÃO LUÍS – MA

2019

* **INTRODUÇÃO**

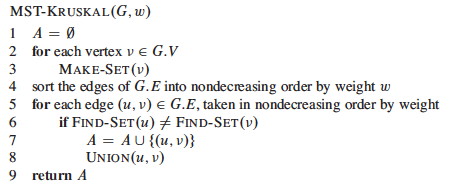
O presente trabalho tem por objetivo implementar os algoritmos de ordenação estudados em sala de aula: InsertSort, SelectSort, ShellSort, QuickSort, MergeSort, HeapSort e CountSort, para serem utilizados na etapa de ordenação das arestas realizadas pelo algoritmo de Kruskal.

O problema da árvore geradora mínima aparece em uma série de aplicações, ou como um subproblema destas. Um exemplo é a instalação de linhas telefônicas (ou elétricas) entre um conjunto de cidades utilizando a infra-estrutura das rodovias com o menor uso de material. Outros problemas (análise de clusters, armazenamento de informações, dentre outros) também podem ser tratados por esta modelagem que possui eficientes algoritmos como Kruskal, Prim e Sollin (Ahuja et. al., 1993).

O algoritmo de Kruskal é um algoritmo em teoria dos grafos que busca uma árvore geradora mínima para um grafo conexo com pesos. Isto significa que ele encontra um subconjunto das arestas que forma uma árvore que inclui todos os vértices, onde o peso total, dado pela soma dos pesos das arestas da árvore, é minimizado. Se o grafo não for conexo, então ele encontra uma floresta geradora mínima (uma árvore geradora mínima para cada componente conexo do grafo).

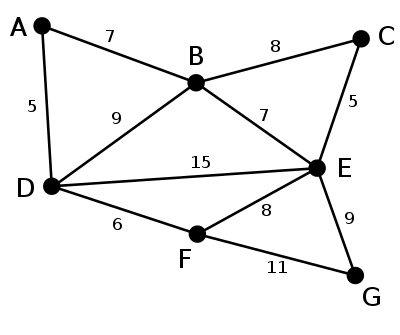
* **KRUSKAL**

O algoritmo de Kruskal foi apresentado por Joseph Kruskal em 1956. Dado um grafo G, deseja-se obter uma árvore geradora mínima. No início, cada vértice de G é considerado uma árvore de um único vértice. A cada iteração do algoritmo, duas árvore são conectadas formando uma árvore maior. A aresta escolhida é aquela que apresenta o menor custo e que nao conecta dois vértices pertencentes à mesma árvore, pois resultaria em um ciclo. O algoritmo é encerrado após todas os vértices estarem presentes na árvore. Este algoritmo tem complexidade O(E log V), onde E ´e o número de arestas no grafo e V ´e o número de vértices, assim como o algoritmo de Prim. Esse algoritmo é um exemplo de um algoritmo guloso (também conhecido como ganancioso ou greedy). Os algoritmos gulosos são utilizado na Teoria dos Grafos para encontrar a árvore geradora mínima de um grafo, que é um subconjunto de arestas formando uma árvore que contém todos os vértices do grafo e cujo peso total das arestas é minimizado. A seguir temos um exemplo de um pseudoalgoritmo:



Uma explicação simplificada do algoritmo de Kruskal temos:

* Etapa 1 - Remover todos os loops e arestas paralelas
* Passo 2 - Organize todas as arestas em ordem crescente de custo
* Passo 3 - Adicione bordas com menos peso



**2.1 Aplicações Reais:**

* **Cabos de aterrissagem**
* **Rede de TV**
* **Operações de excursão**
* **Redes de Computadores**
* **Estudo de ligações moleculares em química**
* **Algoritmos de Roteamento**

1. **ALGORITMOS DE ORDENAÇÃO**

Um tipo de algoritmo muito usado na resolução de problemas computacionais são os [algoritmos de ordenação](http://www.devmedia.com.br/algoritmos-de-ordenacao-em-java/32693), que servem para ordenar/organizar uma lista de números ou palavras de acordo com a sua necessidade. As linguagens de programação já possuem métodos de ordenação, mas é bom saber como funcionam os algoritmos, pois há casos de problemas em que o algoritmo de ordenação genérico não resolve, às vezes é necessário modificá-lo.

Os mais populares algoritmos de ordenação são: *Insertion sort, Selection sort, Bubble sort, Comb sort, Quick sort, Merge sort, Heap sort e Shell sort*. Neste trabalho será feita uma breve análise de número de comparações, número de atribuições e tempo estudados nos algoritmos InsertSort, SelectSort, ShellSort, QuickSort, MergeSort, HeapSort e CountSort, explicando o funcionamento de cada um deles.

* 1. **InsertionSort**

Neste algoritmo de ordenação será eleito o segundo número do vetor para iniciar as comparações. Assim, os elementos à esquerda do número eleito estão sempre ordenados de forma crescente ou decrescente. Logo um laço com as comparações será executado do segundo elemento ao último, ou seja, na quantidade de vezes igual ao número de elementos do vetor menos um. Enquanto existirem elementos à esquerda do número eleito para comparações e a posição que atende a ordenação que se busca não for encontrada, o laço será executado.

* + 1. **Implementação:**

class insertion\_sort(object):

def ordenar(self, colecao):

for i in range(1, len(colecao)):

curNum = colecao[i]

for j in range (i-1,-1,-1):

if colecao[j]['weight']>curNum['weight']:

temp = colecao[j+1]

colecao[j+1] =colecao[j]

colecao[j] = temp

else:

colecao[j+1] = curNum

break

return colecao

* + 1. **Complexidade**

|  |  |  |
| --- | --- | --- |
| Melhor Caso | Caso Médio | Pior Caso |
| O(n) | O(n2) | O(n2) |

* + 1. **Testes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nº Entradas** | **Nº de comparações** | **Nº de Atribuições** | **Tempo** |
| 7 vértices | 35 | 129 | 0.00072s |
| 100 vértices | 14529 | 57639 | 0.00632s |
| 1000 vértices | 2193050 | 8766265 | 0.91721s |

* 1. **SelectionSort**

Este algoritmo é baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o segundo menor valor para a segunda posição e assim sucessivamente, até os últimos dois elementos.

Neste algoritmo de ordenação é escolhido um número a partir do primeiro, este número escolhido é comparado com os números a partir da sua direita, quando encontrado um número menor, o número escolhido ocupa a posição do menor número encontrado. Este número encontrado será o próximo número escolhido, caso não for encontrado nenhum número menor que este escolhido, ele é colocado na posição do primeiro número escolhido, e o próximo número à sua direita vai ser o escolhido para fazer as comparações. É repetido esse processo até que a lista esteja ordenada.

**Neste algoritmo de ordenação cada número do vetor, a partir do primeiro**

* + 1. **Implementação**

class select\_sort(object):

def ordenar(self,colecao):

for i in range (0, len(colecao) - 1) :

minIndex = i

for j in range (i+1, len(colecao)):

if colecao[j]['weight'] < colecao[minIndex]['weight']:

minIndex = j

if minIndex != i:

colecao[i],colecao[minIndex] = colecao[minIndex],colecao[i] (continuação da linha de cima)

return colecao

* + 1. **Complexidade**

|  |  |  |
| --- | --- | --- |
| Melhor Caso | Caso Médio | Pior Caso |
| O(n2) | O(n2) | O(n2) |

* + 1. Testes

|  |  |  |  |
| --- | --- | --- | --- |
| **Nº Entradas** | **Nº de comparações** | **Nº de Atribuições** | **Tempo** |
| 7 vértices | 22 | 141 | 0.00073s |
| 100 vértices | 2138 | 59670 | 0.00967s |
| 1000 vértices | 146877 | 8892036 | 1.50607s |

* 1. **ShellSort:**

É um refinamento do método de inserção direta. O algoritmo difere do método de inserção direta pelo fato de no lugar de considerar o array a ser ordenado como um único segmento, ele considera vários segmentos sendo aplicado o método de inserção direta em cada um deles. Basicamente o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores. Nos grupos menores é aplicado o método da [ordenação por inserção](https://pt.wikipedia.org/wiki/Insertion_sort). Implementações do algoritmo.

* + 1. **Implementação**

class shell\_sort(object):

def ordenar(self,colecao):

h=1

for h in range (1, int(len(colecao)), 3\*h+1):

while (h>=1):

h = int((h-1)/3) #fazer cast não dar prob de float

i=h

while (i<len(colecao)):

aux = colecao[i]

j=i

while (j >= h and int(aux['weight']) < int(colecao[j-h]['weight'])): #(mesma linha de cima)

colecao[j] = colecao[j-h]

j = j-h

colecao[j] = aux

i = i+1

h = h/2

return colecao

* + 1. **Complexidad**e

|  |  |  |
| --- | --- | --- |
| Melhor Caso | Caso Médio | Pior Caso |
| O ( *n* log *n* ) | depende da sequência de lacunas | O(n2) |

* + 1. Testes

|  |  |  |  |
| --- | --- | --- | --- |
| **Nº Entradas** | **Nº de comparações** | **Nº de Atribuições** | **Tempo** |
| 7 vértices | 70 | 256 | 0.00077s |
| 100 vértices | 74145 | 269290 | 0.05412s |
| 1000 vértices | 15858455 | 59102984 | 11.91330s |

* 1. **QuickSort**

O quicksort adota a estratégia de [divisão e conquista](https://pt.wikipedia.org/wiki/Divis%C3%A3o_e_conquista). A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida o quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada

* + 1. **Implementação**

1. Pivô no meio

#(pivo mediana)

class quick\_sort\_mid(object):

def ordenar(self, colecao):

self.sort(colecao, 0, len(colecao) - 1)

return colecao

def sort(self, colecao, left, right):

if left<right:

p = self.partition(colecao, left, right)

self.sort(colecao, left, p-1)

self.sort(colecao, p + 1, right)

def get\_pivot(self, colecao, left, right):

mid = (right + left) // 2

pivot = right

if int(colecao[left]['weight']) < int(colecao[mid]['weight']):

if int(colecao[mid]['weight']) < int(colecao[right]['weight']):

pivot = mid

elif int(colecao[left]['weight']) < int(colecao[right]['weight']):

pivot = left

return pivot

def partition(self, colecao, left, right):

pindex = self.get\_pivot(colecao, left, right)

pvalue = colecao[pindex]

colecao[pindex], colecao[left] = colecao[left], colecao[pindex]

aux = left

for i in range(left, right + 1):

if int(colecao[i]['weight']) < int(pvalue['weight']):

aux += 1

colecao[i], colecao[aux] = colecao[aux], colecao[i]

colecao[left], colecao[aux] = colecao[aux], colecao[left]

return (aux)

**2. Pivô Início:**

class quick\_sort\_beg(object):

def ordenar(self, colecao):

self.sort(colecao, 0, len(colecao) - 1)

return colecao

def sort(self, colecao, left, right):

if left<right:

p = self.partition(colecao, left, right)

self.sort(colecao, left, p-1)

self.sort(colecao, p + 1, right)

def get\_pivot(self, colecao, left, right):

pivot = right

return pivot

def partition(self, colecao, left, right):

pindex = self.get\_pivot(colecao, left, right)

pvalue = colecao[pindex]

colecao[pindex], colecao[left] = colecao[left], colecao[pindex]

aux = left

for i in range(left, right + 1):

if int(colecao[i]['weight']) < int(pvalue['weight']):

aux += 1

colecao[i], colecao[aux] = colecao[aux], colecao[i]

colecao[left], colecao[aux] = colecao[aux], colecao[left]

return (aux)

**3. Pivô Fim:**

class quick\_sort\_end(object):

def ordenar(self, colecao):

self.sort(colecao, 0, len(colecao) - 1)

return colecao

def sort(self, colecao, left, right):

if left<right:

p = self.partition(colecao, left, right)

self.sort(colecao, left, p-1)

self.sort(colecao, p + 1, right)

def get\_pivot(self, colecao, left, right):

pivot = left

return pivot

def partition(self, colecao, left, right):

pindex = self.get\_pivot(colecao, left, right)

pvalue = colecao[pindex]

aux = left

for i in range(left, right + 1):

if int(colecao[i]['weight']) < int(pvalue['weight']):

aux += 1

colecao[i], colecao[aux] = colecao[aux], colecao[i]

colecao[left], colecao[aux] = colecao[aux], colecao[left]

return (aux)

* + 1. **Complexidade**

|  |  |  |
| --- | --- | --- |
| Melhor Caso | Caso Médio | Pior Caso |
| O ( *n* log *n* ) | O ( *n* log *n* ) | O(n2) |

* + 1. Testes (Pivô Início)

|  |  |  |  |
| --- | --- | --- | --- |
| **Nº Entradas** | **Nº de comparações** | **Nº de Atribuições** | **Tempo** |
| 7 vértices | 26 | 120 | 0.00091s |
| 100 vértices | 679 | 3441 | 726 |
| 1000 vértices | 13991 | 59493 | 38.546 |

v. Testes (Pivô Meio)

|  |  |  |  |
| --- | --- | --- | --- |
| **Nº Entradas** | **Nº de comparações** | **Nº de Atribuições** | **Tempo** |
| 7 vértices | 29 | 125 | 0.00120s |
| 100 vértices | 612 | 3458 | 0.01633s |
| 1000 vértices | 11292 | 54196 | 0.42167s |

vi. Testes (Pivô Fim)

|  |  |  |  |
| --- | --- | --- | --- |
| **Nº Entradas** | **Nº de comparações** | **Nº de Atribuições** | **Tempo** |
| 7 vértices | 17 | 83 | 0.00072s |
| 100 vértices | 679 | 2973 | 1.152 |
| 1000 vértices | 12762 | 49966 | 32.383 |

* 1. **HeapSort**

O heapsort utiliza uma estrutura de dados chamada [heap](https://pt.wikipedia.org/wiki/Heap), para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada, lembrando-se sempre de manter a propriedade de max-heap.

A heap pode ser representada como uma árvore (uma árvore binária com propriedades especiais) ou como um vetor. Para uma ordenação decrescente, deve ser construída uma heap mínima (o menor elemento fica na raiz). Para uma ordenação crescente, deve ser construído uma heap máxima (o maior elemento fica na raiz).

* + 1. **Implementação:**

class heap\_sort(object):

def ordenar(self,colecao):

def buil\_max\_heap(colecao):

for i in range (len(colecao)-1,0,-1):

colecao[0], colecao[i] = colecao[i], colecao[0]

self.max\_heapify(colecao,index=0,size=1)

return colecao

def parent(i):

return (i-1)//2

def low(i):

return 2\*1+1

def hi(i):

return 2\*i+2

def buil\_max\_heap(colecao):

length = len(colecao)

start = parent(length-1)

while start >= 0:

max\_heapify(colecao,index= start, size = length)

start = start -1

def max\_heapify(colecao, index, size):

l = low(index)

r = hi(index)

if (l<size and colecao[l]['weight'] >colecao[index]['weight']):

largest = l

else:

largest = index

if (r < size and colecao[r]['weight'] >colecao[largest]['weight']):

largest = r

if (largest != index):

colecao[largest], colecao[index] = colecao[index], colecao[largest]

max\_heapify(colecao, largest, size)

* + 1. **Complexidade**

|  |  |  |
| --- | --- | --- |
| Melhor Caso | Caso Médio | Pior Caso |
| O ( *n* log *n* ) | O ( *n* log *n* ) | O ( *n* log *n* ) |

* + 1. Testes

|  |  |  |  |
| --- | --- | --- | --- |
| **Nº Entradas** | **Nº de comparações** | **Nº de Atribuições** | **Tempo** |
| 7 vértices | 60 | 208 | 0.00063s |
| 100 vértices | 3511 | 10439 | 315 |
| 1000 vértices | 73234 | 205665 | 0.19153s |

* 1. **CountSort**

**Counting sort** é um [algoritmo de ordenação](https://pt.wikipedia.org/wiki/Algoritmo_de_ordena%C3%A7%C3%A3o) estável cuja complexidade é O(n). As chaves podem tomar valores entre 0 e M-1. Se existirem k0 chaves com valor 0, então ocupam as primeiras k0 posições do vetor final: de 0 a k0-1.

A ideia básica do counting sort é determinar, para cada entrada x, o número de elementos menor que x. Essa informação pode ser usada para colocar o elemento x diretamente em sua posição no array de saída. Por exemplo, se há 17 elementos menor que x, então x pertence a posição 18. Esse esquema deve ser ligeiramente modificado quando houver vários elementos com o mesmo valor, uma vez que nós não queremos colocar eles na mesma posição.

**i. Implementação:**

class count\_sort(object):

def ordenar(self,colecao):

tamanho = int(len(colecao))

maior = int(colecao[0]['weight'])

menor = int(colecao[0]['weight'])

for k in colecao:

if int(k['weight']) > maior:

maior = int(k['weight'])

if int(k['weight']) < menor:

menor = int(k['weight'])

media = maior - menor +1

A = [0]\*media

saida =[0]\* tamanho

# passo 1 ----> contagem do número de cada elemento do intervalo

for i in range(0,tamanho):

A[int(colecao[i]['weight'])-menor]+=1;

#passo 2 -----> Agora será feito o complemento de casas de cada valor

for i in range (1,int(len(A))):

A[i] += A[i-1]

#passo 3 --------> alocação dos valores no vetor ordenado.

for i in range(tamanho-1,-1,-1):

saida [A[int(colecao[i]['weight'])-menor]-1] = colecao[i]

A[int(colecao[i]['weight']) - menor]-= 1

#passo 4 --------> passar o conteúdo do vetor final para o vetor original.

for i in range(0,tamanho):

colecao[i] = saida[i]

return colecao

* + 1. **Complexidade**

|  |  |  |
| --- | --- | --- |
| Melhor Caso | Caso Médio | Pior Caso |
| O(n+k) | O(n+k) | O(n+k) |

* + 1. Testes

|  |  |  |  |
| --- | --- | --- | --- |
| **Nº Entradas** | **Nº de comparações** | **Nº de Atribuições** | **Tempo** |
| 7 vértices | 54 | 67 | 0.00091s |
| 100 vértices | 746 | 993 | 0.00443s |
| 1000 vértices | 8962 | 11936 | 0.23598s |

**g. Merge Sort**

O *merge sort*, ou [ordenação](https://pt.wikipedia.org/wiki/Ordena%C3%A7%C3%A3o_(computa%C3%A7%C3%A3o)) por mistura, é um exemplo de [algoritmo de ordenação por comparação](https://pt.wikipedia.org/wiki/Ordena%C3%A7%C3%A3o_por_compara%C3%A7%C3%A3o) do tipo [dividir-para-conquistar](https://pt.wikipedia.org/wiki/Divis%C3%A3o_e_Conquista).

Sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas). Como o algoritmo *Merge Sort* usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

**i. Implementação**

class merge\_sort(object):

def ordenar(self, colecao):

if len(colecao) > 1:

meio = int(len(colecao)/2)

esquerda = colecao[:meio]

direita = colecao[meio:]

self.ordenar(self, esquerda)

self.ordenar(self, direita)

i = j = k = 0

while i < len(esquerda) and j < len(direita):

if int(esquerda[i]['weight']) < int(direita[j]['weight']):

colecao[k] = esquerda[i]

i+=1

else:

colecao[k] = direita[j]

j+=1

k+=1

while i < len(esquerda):

colecao[k] = esquerda[i]

i+=1

k+=1

while j < len(direita):

colecao[k] = direita[j]

j+=1

k+=1

return colecao

* + 1. **Complexidade**

|  |  |  |
| --- | --- | --- |
| Melhor Caso | Caso Médio | Pior Caso |
| O(nlogn) | O(nlogn | O(nlogn |

* + 1. **Testes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nº Entradas** | **Nº de comparações** | **Nº de Atribuições** | **Tempo** |
| 7 vértices | 101 | 177 | 0.00078s |
| 100 vértices | 5.373 | 7.278 | 0.05412s |
| 1000 vértices | 98.537 | 121.356 | 11.91410s |

1. **CONCLUSÃO**

Analisando os gráficos tendo como parâmetros a quantidade de entrada de dados: 7, 100 e 1000 obtemos os seguintes resultados:

* InsertSort:

É benéfico para pequenas estruturas lineares, quando testado com 7 vértices, resulta em um tempo aceitável.

* SelectSort:

Apresenta a mais simples implementação de todas, se mostrando um algoritmo bem intuitivo, no entanto, seus resultados se mostram bem ruins se comparados com outros, principalmente se levarmos em consideração o tempo de espera que se mostra bem grande. No entanto, percebemos que se torna bastante aceitável se testado com 7 vértices, semelhante ao InsertSort, especialmente quando a lista se mostra quase totalmente ordenada

* ShellSort:

Devido a sua complexidade possui excelentes desempenhos em N muito grandes, inclusive sendo melhor que o Merge Sort.

* HeapSort:

Tem um desempenho em tempo de execução muito bom em conjuntos ordenados aleatoriamente, mostrando um uso de memória bem comportado e o seu desempenho em [pior cenário](https://pt.wikipedia.org/wiki/Complexidade_Pior_caso) é praticamente igual ao desempenho em [cenário médio](https://pt.wikipedia.org/wiki/Complexidade_Caso_m%C3%A9dio).

* MergeSort:

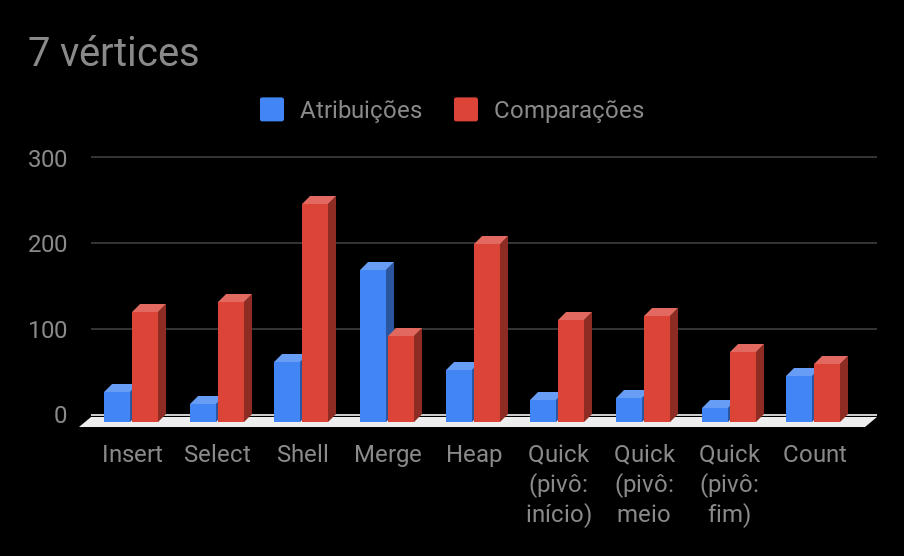
Mostrou um desempenho mediano, essencialmente nas comparações, sua recursividade garante bons resultados quando tem que lidar com listas aleatórias, entretanto, ao contrário do insert e select, que lidam bem com estruturas semi-ordenadas, o merge acaba por realizar comparações e atribuições desnecessárias

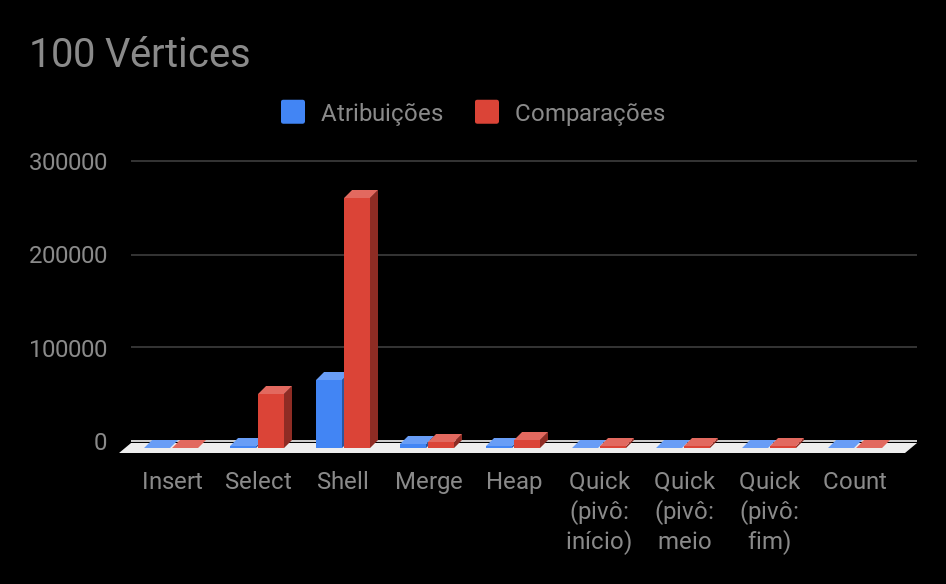
* QuickSort:

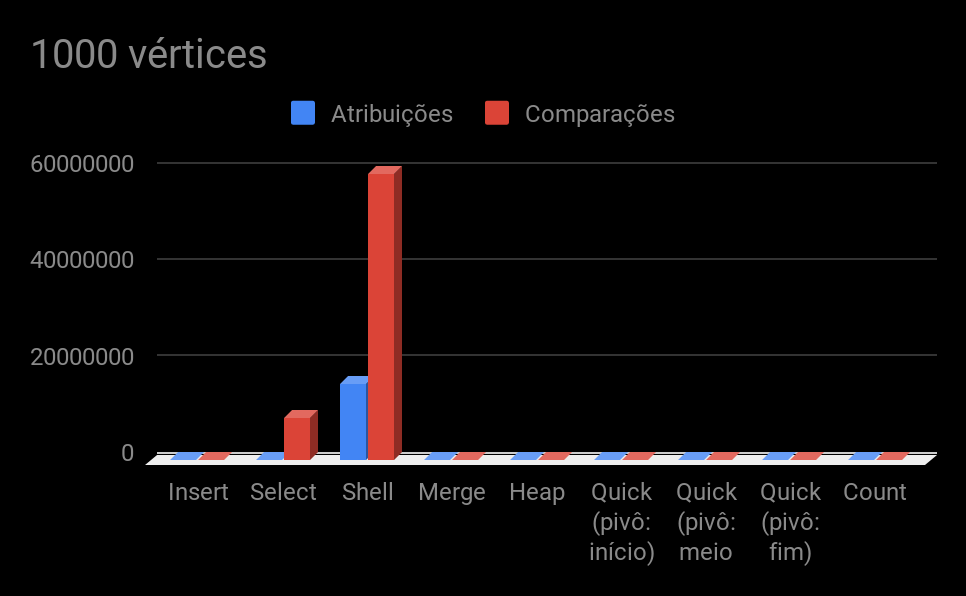
Mostrou um desempenho mediano, tem seu tempo de execução reduzido, todavia, possui alta quantidade de atribuições e comparações. Mostra-se uma boa escolha para execução em menor tempo, possuindo como bom atributo a não necessidade de memória auxiliar para realizar sua tarefa com êxito.

* CountSort:

Apresentou o melhor desempenho dentre todos, possuindo a necessidade de memória auxiliar.

****

****

****

**REFERÊNCIAS**

CORMEN, Thomas H. et al. **Introduction to algorithms**. MIT press, 2009.