**RELATÓRIO ESTRUTURA DE DADOS 2 (ATIVIDADE PRÁTICA 1)**

**ALUNO: FELIPE EDUARDO AGUIAR OLIVEIRA**

* **Declarações iniciais**

Os algoritmos de ordenação implementados nesta atividade serviram na etapa de ordenação das arestas realizada pelo algoritmo de Kruskal. Neste relatório contém explicações sobre cada algoritmo e seus funcionamentos, assim como resultados dos testes realizados com as vértices disponibilizadas nos Datasets.

* **Especificação da máquina utilizada**

Notebook Samsung

Intel(R) Core(TM) i3 – 6006U CPU @ 2.000GHz 1.99GHz

Memória RAM de 4,00 GB

Sistema operacional de 64 bits, processador com base em x64

* **QuickSort**

O Quicksort é um dos algoritmos de ordenação mais conhecidos. Por ser um dos mais eficientes é largamente utilizado/customizado em projetos de linguagens de programação. Ele pertence a categoria de Ordenação por Troca ou Exchange Sort. O algoritmo utiliza o paradigma de programação Dividir para Conquistar. Esse paradigma é uma abordagem recursiva em que a entrada do algoritmo é ramificada múltiplas vezes a fim de quebrar o problema maior em problema menores da mesma natureza. A seguir detalhamento do código implementado e comentado.

class QuickSort(object):

def ordenar(self, colecao):

**#VERIFICA SE A COLEÇÃO E INICIA O METODO "TESTE" COM SEU INICIO (LOW) E FIM (HIGH)**

if colecao is not None:

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

return colecao

**#METODO RECURSIVO PARA ORDENAR A COLEÇÃO, SEJA ELA DIVIDIDA OU NÃO, NO CASO DA PRIMEIRA CHAMADA**

def teste(self, colecao, low, high):

if low < high:

pi = self.partition(colecao, low, high)

self.teste(colecao, low, pi-1) **#DIVIDE A COLEÇÃO PELA METADE**

self.teste(colecao, pi+1, high) **#DIVIDE A COLEÇÃO PELA METADE**

return colecao

**#RECEBE A COLEÇÃO DIVIDIDA OU NÃO, NO CASO DA PRIMEIRA CHAMADA**

def partition(self, colecao, low, high):

i = (low-1)

pivot = colecao[high] **#O ÚLTIMO FOI ESCOLHIDO COMO PIVÔ**

for j in range(low, high):

if int(colecao[j]['weight']) <= int(pivot['weight']):

i = i + 1

colecao[i], colecao[j] = colecao[j], colecao[i]

colecao[i+1], colecao[high] = colecao[high], colecao[i+1]

return (i+1)

Dado a sequência de entrada, o método particionador deve primeiramente escolher um elemento que chamaremos de pivô. Em seguida iterar sobre toda a coleção a fim de posicionar todos elementos menores do que esse pivô à sua esquerda. A escolha do pivô pode ser feita aleatoriamente, ser o primeiro elemento ou o último. Na minha implementação utilizei o último elemento da coleção.

* Análise do QuickSort:

Uma imagem contendo objeto

Descrição gerada automaticamenteO tempo gasto pelo QuickSort em geral pode ser escrito da seguinte maneira:

Os dois últimos termos são para duas chamadas recursivas, o primeiro termo é para o processo de partição. O k é número de elementos menores que o pivô. O tempo gasto pelo QuickSort depende da entrada e da estratégia de partição. A seguir estão dois casos.

**Pior caso:** O pior caso ocorre quando o processo de partição sempre escolhe o elemento maior ou menor como pivô. Se considerarmos a estratégia de partição acima, na qual o último elemento é sempre escolhido como pivô, o pior caso ocorrerá quando a coleção já estiver classificada em ordem crescente ou decrescente. A seguir, é recorrente o pior caso.

Uma imagem contendo céu

Descrição gerada automaticamente



A solução da recorrência acima é (n 2).

**Melhor caso:** o melhor caso ocorre quando o processo de partição sempre escolhe o elemento do meio como pivô. A seguir, é recorrente o melhor caso.

A solução da recorrência acima é (teta)(nLogn). Pode ser resolvido usando o caso 2 do [Teorema Mestre](http://en.wikipedia.org/wiki/Master_theorem).

* Resultados dos testes realizados

Ao executar o algoritmo de ordenação quickSort para ordenar a coleção recebida ele demonstrou respostas rápidas até a quantidade 10 mil de vértices, apresentando os seguintes resultados:

7 vértices: entre 0,2 e 0,3 segundos

100 vértices: entre 0,2 e 0,3 segundos

1000 vértices: entre 0,5 e 0,8 segundos

10000 vértices: entre 60 e 62,2 segundos

Todos estes executados em meu notebook (especificação no tópico de arquitetura da máquina utilizada). Porém, me deparei com uma demora com a quantidade 100000 vértices que não encerrava a execução.

- Em relação ao número de comparações criei uma contagem (variável) dentro da função “partition”, detalhando cada número de comparações com suas chamadas. Abaixo contém a quantidade para cada vértice:

7 vértices: 18 comparações

100 vértices: 4309 comparações

1000 vértices: 13505 comparações

10000 vértices: 1213505 comparações

Não obtive êxito em relação a quantidade de 100000 vértices. Como não encerrava a execução, o número de comparações não era esclarecido.

* **MergeSort**

A ideia do MergeSort é dividir o vetor em dois sub - vetores, cada um com metade dos elementos do vetor original. Esse procedimento é então reaplicado aos dois sub - vetores recursivamente. Quando os sub - vetores têm apenas um elemento (caso base), a recursão para. Então, os sub - vetores ordenados são fundidos (ou intercalados) num único vetor ordenado. Abaixo é apresentado o código implementado e comentado.

class MergeSort(object):

def ordenar(self, colecao):

if len(colecao)>1:

mid = int(len(colecao))//2 **#divide o tamanho da colecao recebida em dois**

L = colecao[:mid] **#pega o lado esquerdo da colecao**

R = colecao[mid:] **#pega o lado direito da colecao**

self.ordenar(L) **#ordena lado esquerdo**

self.ordenar(R) **#ordena lado direito**

i = j = k = 0

**#compara as sub colecao**

while i < len(L) and j < len(R):

if int(L[i]['weight']) < int(R[j]['weight']):

colecao[k] = L[i]

i = i + 1

else:

colecao[k] = R[j]

j = j + 1

k = k + 1

while i < len(L):

colecao[k] = L[i]

i = i + 1

k = k + 1

while j < len(R):

colecao[k] = R[j]

j = j + 1

k = k + 1

return coleção

* Análise do MergeSort

O MergeSort é um algoritmo recursivo e a complexidade do tempo pode ser expressa da seguinte forma:

Uma imagem contendo objeto

Descrição gerada automaticamente

A recorrência acima pode ser resolvida usando o método mestre, caos 2, e a solução da recorrência é

A complexidade de tempo da MergeSort acima ocorre nos três casos (pior, médio e melhor), pois a classificação de mesclagem sempre divide a coleção em duas metades e leva um tempo linear para mesclar duas metades.

* Resultado dos testes realizados

Ao executar o algoritmo de ordenação MergeSort para ordenar a coleção recebida ele demonstrou respostas rápidas até a quantidade 10 mil de vértices (assim como no quickSort), apresentando os seguintes resultados:

7 vértices: entre 0,1 e 0,2 segundos

100 vértices: entre 0,1 e 0,2 segundos

1000 vértices: entre 0,5 e 0,8 segundos

10000 vértices: entre 40 e 42,2 segundos

Ao realizar no tamanho de 100000 vértices, a execução em meu notebook encerrou após uma hora de ordenação.

- Assim como no QuickSort criei um contador na função para explorar a quantidade de comparações. Abaixo estão os valores:

7 vértices: 13 comparações

100 vértices: 353 comparações

1000 vértices: 4432 comparações

10000 vértices: 44760 comparações

Houve certa impossibilidade ao calcular o número de comparações da quantidade de 100000 vértices.

* **InsertSort**

O Insertion sort é um algoritmo simples e eficiente quando aplicado em pequenas listas. Neste algoritmo a lista é percorrida da esquerda para a direita, à medida que avança vai deixando os elementos mais à esquerda ordenados. O algoritmo funciona da mesma forma que as pessoas usam para ordenar cartas em um jogo de baralho como o pôquer.

class InsertionSort(object):

def ordenar(self, colecao):

for i in range(1, len(colecao)): **# COMEÇA A PERCORRER A PARTIR DO SEGUNDO**

key = colecao[i]

j = i - 1

while j>=0 and int(key['weight']) < int(colecao[j]['weight']): **# LAÇO PARA VOLTAR AS POSIÇÕES DOS DADOS COMPARADOS DA COLEÇÃO**

colecao[j+1] = colecao[j]

j = j - 1

colecao[j+1] = key

return coleção

* Análise do Insertsort

**Melhor caso:** O(n), quando a coleção está ordenada.

**Pior caso:** O(n²), quando a coleção está em ordem inversa, daquela que deseja ordenar.

* Resultado dos testes realizados

Ao executar o algoritmo de ordenação InsertSort para ordenar a coleção recebida ele demonstrou respostas rápidas até a quantidade 10 mil de vértices (assim como no quickSort e MergeSort), apresentando os seguintes resultados:

7 vértices: entre 0,1 e 0,2 segundos

100 vértices: entre 0,3 e 0,4 segundos

1000 vértices: entre 2,5 e 2,8 segundos

10000 vértices: entre 5 e 6 minutos

Ao realizar o teste com 100000 vértices não foi obtido êxito em sua execução e resultado.

- Testes realizados para obter o número de comparações:

7 vértices: 10 comparações

100 vértices: 243 comparações

1000 vértices: 2969 comparações

10000 vértices: 30023 comparações

Como o código não obteve êxito com 100000 vértices, não foi possível obter o total de comparações.

O arquivo destinado a criação deste trabalho traz a seguinte pergunta: **“Discuta no relatório os resultados obtidos por essas abordagens. A substituição do algoritmo de**

**Inserção pelos algoritmos SelectSort e ShellSort resultaria em redução no tempo de ordenação?”**

Realizando os testes com esses dois algoritmos, obtive os seguintes resultado:

* **SelectionSort**

Sua ideia consiste em ordenar a lista “selecionando” a cada iteração os menores itens possíveis e os colocam da esquerda para a direita.

Código implementado e comentado:

class SelectionSort(object):

def ordenar(self, colecao):

for i in range(len(colecao)):

min\_idx = i **# SELECIONA O PRIMEIRO DADO DA COLEÇÃO COMO MENOR**

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

if int(colecao[min\_idx]['weight']) > int(colecao[j]['weight']): **# LAÇO PARA PERCORRER A COLEÇÃO E COMPARAR PARA ACHAR UM MENOR**

min\_idx = j

colecao[i], colecao[min\_idx] = colecao[min\_idx], colecao[i]

return coleção

Na execução do código com os mesmos dados dos vértices:

7 vértices: entre 0,2 e 0,3 segundos

100 vértices: entre 0,3 e 0,4 segundos

1000 vértices: entre 4,5 e 5 segundos

10000 vértices: entre 7 e 7,55 minutos

Ao executar o código com 100000 vértices não foi obtido êxito nos resultados, pois ele não terminava a execução. Pode – se concluir dai o pior caso do algoritmo que O(n2)

* **ShellSort**

Ordenação por inserção só troca itens adjacentes para determinar o ponto de inserção. Nesse algoritmo são efetuadas n-1 comparações e movimentações quando o menor item está na última posição. O método de Shell contorna este problema permitindo trocas de registros distantes.

Código implementado:

class ShellSort(object):

def ordenar(self, colecao):

n = len(colecao)

gap = n//2

while gap > 0:

for i in range(gap, n):

temp = colecao[i]

j=i

while j >= gap and int(colecao[j-gap]['weight']) > int(temp['weight']):

colecao[j] = colecao[j-gap]

j = j - gap

colecao[j] = temp

gap//=2

return coleção

Na execução do código com os mesmos dados dos vértices:

7 vértices: entre 0,2 e 0,3 segundos

100 vértices: 0,3 segundos

1000 vértices: entre 0,8 e 0,9 segundos

10000 vértices: entre 34,3 e 35 segundos

Não foi obtido êxito na execução do código para 100000 vértices provando o pior caso do algoritmo que é O(n2).

* **Referências**

<https://www.blogcyberini.com/2018/07/merge-sort.html>

<https://blog.pantuza.com/artigos/o-algoritmo-de-ordenacao-quicksort>

<https://www.geeksforgeeks.org/shellsort/>