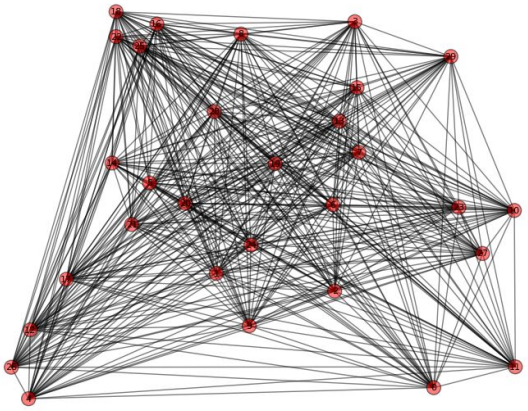
O PROBLEMA DO CAIXEIRO VIAJANTE



Com o objetivo de implementar um programa que deve ler um grafo Hamiltoniano ponderado a partir de um arquivo qualquer e através de um algoritmo visto em sala (2-otimal ou Twice-Around) obter 10 soluções diferentes para o problema do caixeiro-viajante.

METODOLOGIA

Para obter soluções distintas para o problema há algumas heurísticas comumente adotadas na prática: utilizar diferentes inicializações, ou seja, soluções iniciais. Elas podem ser geradas simplesmente aleatoriamente (selecionando vértices quaisquer) ou utilizando alguma heurística, como por exemplo a escolha do vizinho mais próximo por exemplo. Dessa forma, escolhe-se aleatoriamente apenas o primeiro vértice do ciclo (v0) e depois sempre é escolhido como próximo elemento da sequência o vizinho mais próximo do vértice atual, até que o ciclo Hamiltoniano seja formado (não sobre mais vértices).

Dadas as condições anteriores, foi implementado uma solução utilizando Python e as bibliotecas NetworkX, Numpy e Matplotlib.

A solução ficou da seguinte forma:

Algoritmo:

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
from heapq import heappush
from heapq import heappop
Tam = 30

def twiceAround(G, ini = 0):
    F = nx.minimum_spanning_tree(G) #Gera uma MST do grafo G
    F = nx.MultiGraph(F) #Define um MultiGrafo

    for u,v in list(F.edges()): #Dublica aresta da mst
        F.add_edge(u,v)

    #Gera o caminho de euler
    euler= list(nx.eulerian_circuit(F, ini))
    #Inicializa o grafo e cria um auxiliar
    caminho = nx.Graph()
    Aux = []
    #Salva o ciclo Euleriano no Aux
    for u,v in euler:
        Aux.append(u)
        Aux.append(v)

    fila = []


    for i in Aux:
        #Elimina as repetições
        if(i not in fila):
            fila.append(i)
    fila.append(ini)
    for i in range(Tam):
        #Grafo resultante
        caminho.add_edge(fila[i], fila[i+1])
        #Copia os pesos
        caminho[fila[i]][fila[i+1]]['weight'] = G[fila[i]][fila[i+1]]['weight']
    return caminho
```

```
#Função para registrar o custo de cada caminho
def calc_custo(G):
    custo = 0
    custos = nx.get_edge_attributes(G, 'weight')
    for v in G.edges():
        custo = custo + custos[v]
    return custo

A = np.loadtxt("ha30_dist.txt")
G = nx.from_numpy_matrix(A)

#Lista dos custos de cada caminho
caminhos = []

for i in range(30):
    Caminho = twiceAround(G, i)
    custo = calc_custo(Caminho)
    heappush(caminhos, (custo, i))

for i in range(30):
    custo, inicial = heappop(caminhos)
    print("Caminho inicia em ", inicial, " com custo: ", custo)


nx.draw(Caminho, with_labels=True)
plt.show()
```

Questionamento:

Foi proposto também achar as 3 melhores e as 3 piores soluções, assim como qual a diferença de custo entre a melhor e a pior?

A partir do algoritmo a cima, obteve os seguintes resultados:

Os 3 melhores:

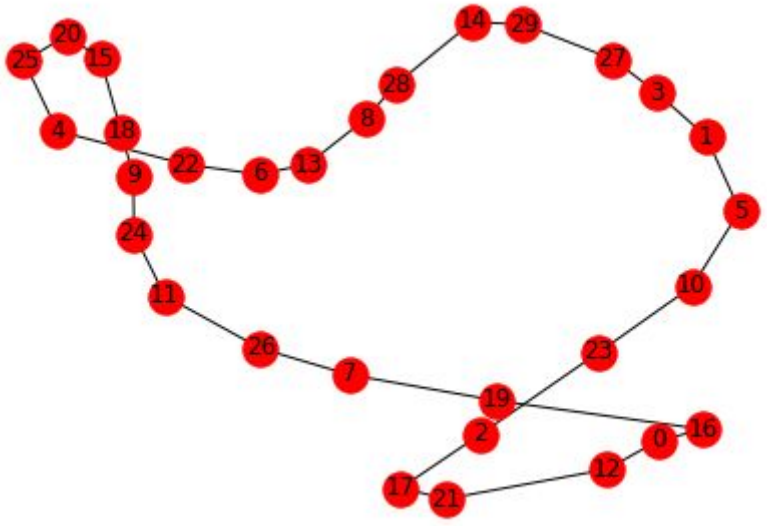
Caminho inicia em 9 com custo: 567.0_
Caminho inicia em 27 com custo: 570.0_
Caminho inicia em 29 com custo: 570.0_

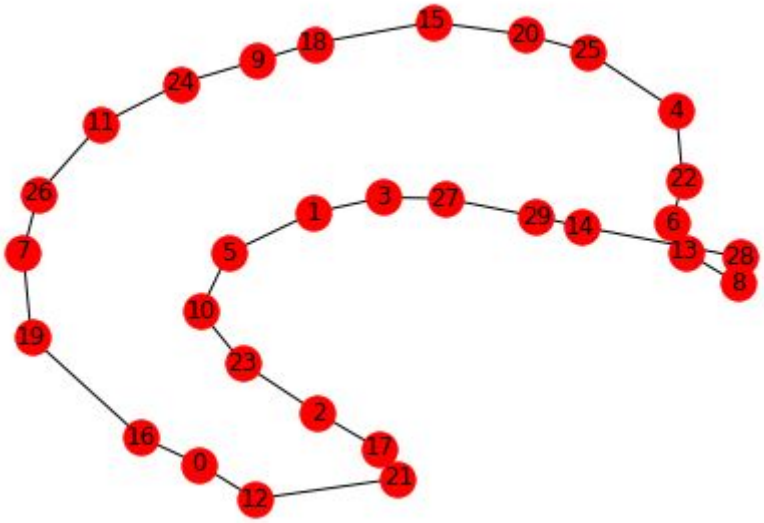
Os 3 piores:

Caminho inicia em 22 com custo: 648.0_
Caminho inicia em 4 com custo: 658.0_
Caminho inicia em 25 com custo: 658.0_

A diferença entre o melhor caminho (Iniciado em 9) e o pior caminho (Iniciado em 25) foi de 91. Uma diferença de aproximadamente 16,05% entre o melhor e o pior caso.

Imagem gerada:




15
20
25
18
9
24
4
11
22
26
1
3
27
29
6
14
13
28
7
5
8
10
19
23
2
16
17
0
21
12