GCC 218 - Algoritmos em Grafos - Trabalho Final

Alunos:
Héuller Silva
João Pedro Andolpho
Luiz Carlos Conde
Gabriel Amorim
Renan Modenese
Victor Landin

Professor:
Mayron César de O. Moreira

Dezembro de 2018

Introdução

Para implementar a solução proposta, inicialmente é feita a importação dos módulos necessários, declaração de classes e leitura do arquivo contendo uma instância de teste.

In [1]:
```
import networkx as nx
from networkx import MultiGraph
import matplotlib.pyplot as plt
from math import sqrt
from statistics import mean, stdev
import random

# Instancia grafo
G = nx.MultiGraph()
```

As classes a seguir são utilizadas na etapa de roteirização. A etapa de subdivisão utiliza apenas a classe Multigraph, parte do módulo NetworkX.

In [2]:
```
class Veiculo:
    def __init__(self, V, P, Nv, vf, vd, tc, td, ph, pkm, pf, tipo):
        self.volume_max = V
        self.valor_max = P
        self.quantidade = Nv
        self.velocidade_centro = random.randint(vf - 5, vf + 5)
        self.velocidade = random.randint(vd - 5, vd + 5)
        self.tempo_carga = random.uniform(tc, 3*tc)
        self.tempo_descarga = td
        self.custo_hora = ph
        self.custo_km = pkm
        self.custo_fixo = pf
        self.tipo = tipo
```

A leitura do arquivo é feita pelo código a seguir

In [3]:
```
with open('InstanciaTeste.txt') as arquivo:
    # le o arquivo, linha a linha
    numero_clientes = int(arquivo.readline())
    numero_regioes = int(arquivo.readline())
    tipo_veiculos = int(arquivo.readline())
    carga_horaria = int(arquivo.readline())
    # 5 primeiros vértices são centros de distribuição
    for i in range(numero_regioes):
        centro_dist = arquivo.readline()
        dados_centro = centro_dist.split()
        x, y = [float(valor) for valor in dados_centro[:2]]
        G.add_node(i, x=x, y=y, volume=0, valor=0)
    # demais vértices são clientes
    for i in range(numero_regioes, numero_clientes):
        cliente = arquivo.readline()
        dados_cliente = cliente.split()
        x, y, v = [float(valor) for valor in dados_cliente[:3]]
        p, n = [int(valor) for valor in dados_cliente[3:]]
        G.add_node(i, x=x, y=y, volume=v, valor=p, quantidade=n)
    # instancia e lê os dados dos veiculos
    veiculos = {}
    for tipo in ('van', 'minivan', 'carro', 'moto', 'terceirizado'):
        veiculo = arquivo.readline()
        dados_veiculo = veiculo.split()
        V = float(dados_veiculo[0])
        P, Nv, vf, vd = [int(valor) for valor in dados_veiculo[1:5]]
        tc, td = [float(valor) for valor in dados_veiculo[5:7]]
        ph, pkm, pf = [int(valor) for valor in dados_veiculo[7:]]
        veiculos[tipo] = Veiculo(V, P, Nv, vf, vd, tc, td, ph, pkm, pf, tipo)
# fim do arquivo
```

Finalmente é declarada a função responsável pelo desenho do grafo.

In [4]:
```
def desenha_grafo(G, tamanho=(5, 5), resolucao=150):
    # tamanho em polegadas, deve ter proporção 1:1
    plt.figure(figsize=tamanho, dpi=resolucao)
    # gera dicionario com tupla de coordenadas para cada vértice
    posicao = {}
    for v in G.nodes():
        posicao[v] = tuple(G.nodes.data()[v][k] for k in ('x', 'y'))
    # gera lista de cores de acordo com a região dos vértices
    cores = ('c', 'm', 'y', 'r', 'g')
    lista_cores = [cores[G.nodes.data()[v]['regiao']] if v > 4
                   else 'k' for v in G.nodes()]
    # desenho de G
    nx.draw(G, pos=posicao, with_labels=True, font_size=8, font_color='w',
            font_weight='bold', node_size=100, node_color=lista_cores)
    plt.show()
```

Etapa 1: subdivisão

Para a solução da subdivisão do grafo, foram realizadas diversas tentativas com resultados bastante divergentes. Um breve histórico da solução:

- **Distribuição por centro mais próximo:** Inicialmente fizemos a distribuição dos vértices para as regiões cujo centro está mais próximo. A solução obtida, embora satisfatória em termos de distâncias, gerava regiões com cardinalidades, e principalmente demandas bastante heterogêneas.
- **Distribuição alternada entre regiões:** Também utiliza o critério da distância dos centros, porém de forma alternada entre as regiões, adicionando um vértice por região em cada iteração. Com isso, ainda que as demandas continuem heterogêneas, a cardinalidade das regiões era uniforme.

  Foram feitas diversas tentativas de melhorar o resultado inicial por meio de trocas, mas sem melhora safisfatória em termos de demandas, e comprometendo a uniformidade da cardinalidade.
- **Distribuição por região de menor demanda:** A última solução obtida também considera as distâncias dos centros ao distribuir os vértices, porém a distribuição é feita sempre na região com menor demanda. Como inicialmente todas as demandas são igual a 0, e aumentam a medida que adicionamos vértices à região, isso faz com que a distribuição ainda ocorra de forma alternada, e ainda garante que as demandas regionais sejam uniformes.

  Como desvantagem, os últimos vértices adicionados a região tendem a ficar isolados dos demais. Realizamos uma modificação nessa solução, de forma a evitar a ocorrência de vértices isolados, ainda que isso tenha um impacto considerável da uniformidade das demandas. Ambas as soluções foram inclusas, para fins de comparação.

Funções auxiliares:

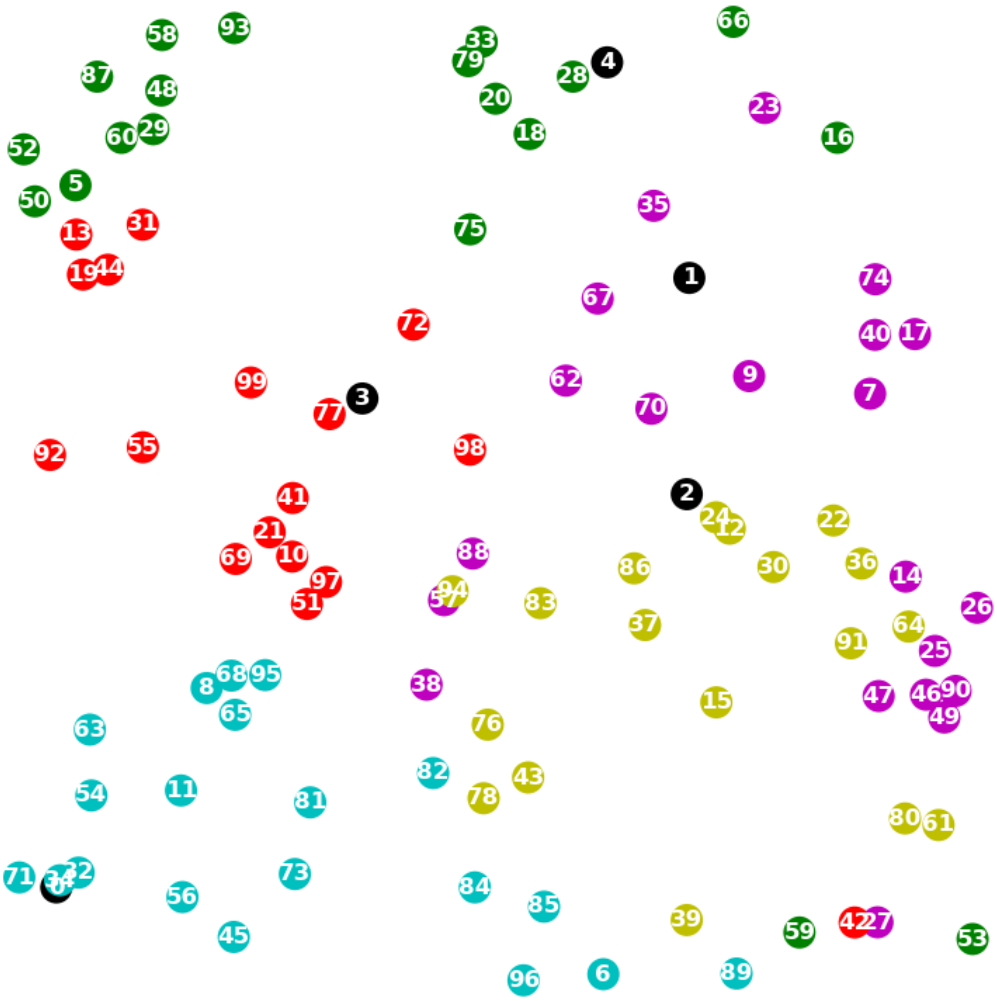
In [5]:
```
def calcula_distancia(G, u, v):
    """Calcula a distância euclideana entre os vértices u,v de G."""
    distancia = sqrt((G.nodes[u]['x'] - G.nodes[v]['x'])**2 +
                     (G.nodes[u]['y'] - G.nodes[v]['y'])**2)
    return distancia

def demanda_vertice(G, v):
    """Calcula a demanda de um cliente, considerando volume e valor do pedid
o"""
    return G.nodes[v].get('volume') * G.nodes[v].get('valor')

def obter_vertice(G, nao_alocados, r):
    """Retorna o vértice não alocado mais próximo do centro da região R"""
    proximo = next(iter(nao_alocados))
    for v in nao_alocados:
        if (calcula_distancia(G, r, v) < calcula_distancia(G, r, proximo)):
            proximo = v
    return proximo
```

Finalmente, temos a heurística de distribuição:

In [6]:
```python
def subdivisao(G, k):
    """Heurística para distribuição dos vértices em regiões, priorizando
    as regiões com menor demanda.

    args:
        G: grafo não direcionado onde os k primeiros vértices são centros
        k: número de regiões desejadas

    returns:
        regioes: lista de conjuntos contendo os vértices de cada região
    """

    # inicializa as regiões com seus centros
    regioes = [{centro} for centro in range(k)]

    # inicializa variáveis
    nao_alocados = {v for v in G.nodes() if v >= k}
    demandas = [0 for r in range(k)]

    # distribuição inicial
    while nao_alocados:
        # encontra região com menor demanda
        r = demandas.index(min(demandas))
        # obtem vértice mais próximo do centro de r
        v = obter_vertice(G, nao_alocados, r)
        # soma demanda do vértice a demanda da região
        demandas[r] += demanda_vertice(G, v)
        # aloca vertice na regiao r e remove dos não alocados
        regioes[r].add(v)
        G.nodes()[v]['regiao'] = r
        nao_alocados -= {v}

    # Calcula a demanda média e os desvios
    demanda_ideal = sum(demandas) / k
    desvios = [stdev([demandas[r], demanda_ideal]) for r in range(k)]

    # Imprime os resultados obtidos
    print('Demandas: {}'.format(demandas))
    print('Demanda ideal: {}'.format(demanda_ideal))
    print('Desvios: {}'.format(desvios))
    print('Soma dos desvios: {}'.format(sum(desvios)))

    return regioes
```

In [7]:
```
# Subdivisão e desenho do grafo
subdivisao(G, numero_regioes)
desenha_grafo(G)
```

```
Demandas: [55.406663440544655, 56.845535611212505, 57.30087640159126, 54.35070
561317864, 51.33754200547936]
Demanda ideal: 55.04826461440128
Desvios: [0.2534262403352766, 1.2708625094751205, 1.592837070102781, 0.4932487
000422463, 2.6238771198709316]
Soma dos desvios: 6.234251639826356
```



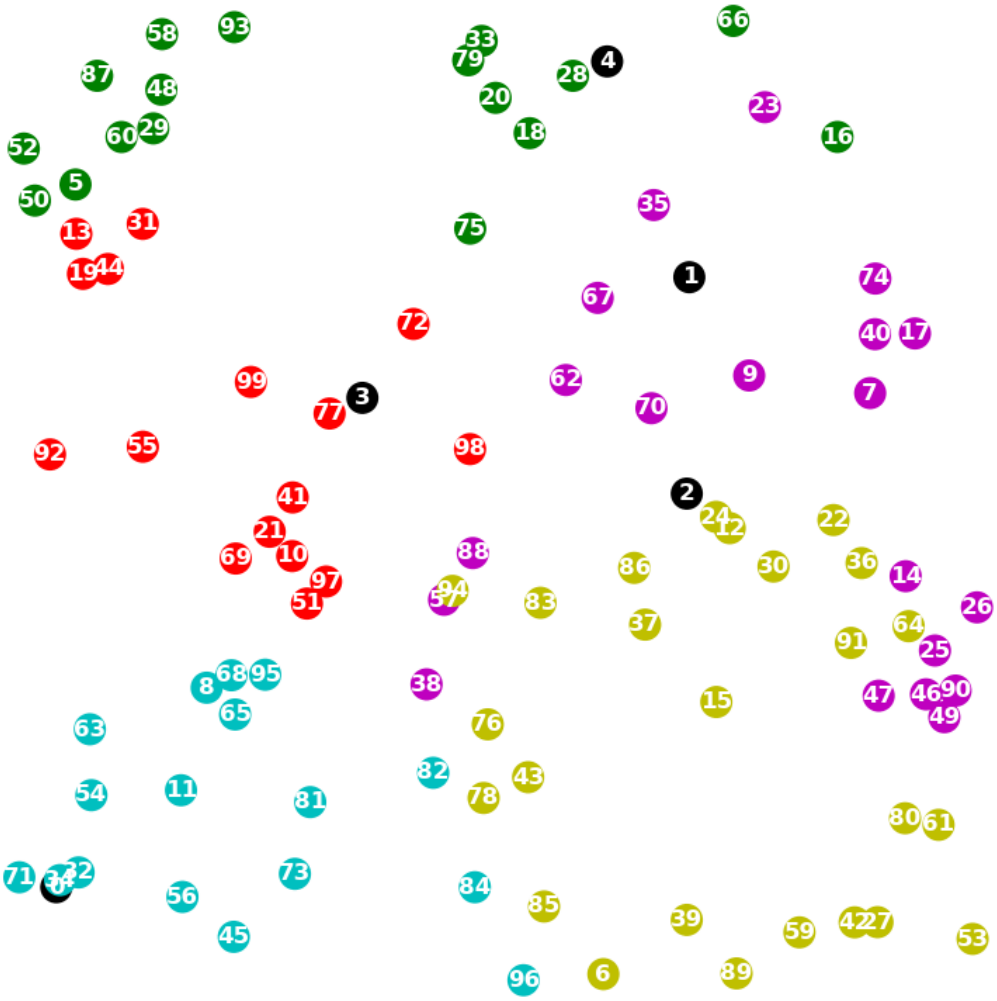
A heurística modificada, que minimiza a ocorrência de vertices isolados:

In [8]:
```
def obter_vertice_alt(G, k, matrizes_ordenadas, nao_alocados, r):
    matriz_distancias = matrizes_ordenadas[r]
    for u in range(G.order() - k):
        v = matriz_distancias[u][0]
        if v in nao_alocados:
            return v

def subdivisao_alt(G, k):
    """Heurística para distribuição dos vértices em regiões, priorizando
    as regiões com menor demanda, modificada para minimizar a ocorrência de
    vértices isolados.

    args:
        G: grafo não direcionado onde os k primeiros vértices são centros
        k: número de regiões desejadas

    returns:
        regioes: lista de conjuntos contendo os vértices de cada região
    """

    # inicializa as regiões com seus centros
    regioes = [{centro} for centro in range(k)]

    # calcula distancia dos vértices para cada um dos centros
    matriz_distancias = {}
    for v in range(k, G.order()):
        distancias = [calcula_distancia(G, centro, v) for centro in range(k)]
        matriz_distancias[v] = distancias

    # ordena matriz_distancias para cada região, salvando resultados numa lista
    matrizes_ordenadas = [sorted(matriz_distancias.items(),
                          key=lambda v: v[1][r]) for r in range(k)]

    # inicializa variáveis
    nao_alocados = {v for v in G.nodes() if v >= k}
    demandas = [0 for r in range(k)]
    # os n últimos vértices são distribuídos separadamente
    n = len(nao_alocados) / 10

    while len(nao_alocados) > n:
        # encontra região com menor demanda
        r = demandas.index(min(demandas))
        # obtem vértice mais próximo do centro de r
        v = obter_vertice_alt(G, k, matrizes_ordenadas, nao_alocados, r)
        # soma demanda do vértice a demanda da região
        demandas[r] += demanda_vertice(G, v)
        # aloca vertice na regiao r e remove dos não alocados
        regioes[r].add(v)
        G.nodes()[v]['regiao'] = r
        nao_alocados -= {v}

    # distribui os vértices restantes considerando apenas o centro mais próximo
    while nao_alocados:
        for v in list(nao_alocados):
            r = matriz_distancias[v].index(min(matriz_distancias[v]))
            demandas[r] += demanda_vertice(G, v)
            regioes[r].add(v)
            G.nodes()[v]['regiao'] = r
            nao_alocados -= {v}

    # Calcula a demanda média e os desvios
    demanda_ideal = sum(demandas) / k
    desvios = [stdev([demandas[r], demanda_ideal]) for r in range(k)]

    # Imprime os resultados obtidos
    print('Demandas: {}'.format(demandas))
    print('Demanda ideal: {}'.format(demanda_ideal))
```

```
    print('Desvios: {}'.format(desvios))
    print('Soma dos desvios: {}'.format(sum(desvios)))

    return regioes
```

In [9]:
```
# Subdivisão e desenho do grafo
regioes = subdivisao_alt(G, numero_regioes)
desenha_grafo(G)
```

```
Demandas: [48.77509220546873, 50.98991274350063, 75.55350880314685, 50.5371513
6206695, 49.385657957823256]
Demanda ideal: 55.04826461440128
Desvios: [4.435802749908558, 2.8696881283549622, 14.49939721574804, 3.18983877
1426108, 4.004067566058407]
Soma dos desvios: 28.998794431496073
```



Etapa 2: roteirização

Para a etapa de roteirização, além da classe Veiculo descrita anteriormente, foram usadas outras duas classes, descritas a seguir.

- **Classe Entrega:**

  Armazena dos dados de cada entregador.

  A função `prossegue` é responsavel por verificar se adicionar um cliente a uma determinada rota é seguro. Para verificar isto calcula-se se adicionar o cliente não irá exceder nenhum limitante da entrega ou veiculo, como a capacidade do veiculo, valor máximo que o veiculo pode transportar ou carga horária de trabalho.

In [30]:
```
class Entrega:

    def __init__(self, carga_horaria):
        self.tempo_restante = carga_horaria
        self.pacotes = 0
        self.volume = 0
        self.valor = 0
        self.tipo_veiculo = ''
        self.veiculo = None
        self.rota = []
        self.km = 0

    def prossegue(self, Gr, matriz, u, v, r, entrega, veiculo):
        tempo = (matriz[u][v] / veiculo.velocidade + matriz[v][r] / veiculo.vel
ocidade_centro
                 + Gr.node[v].get('quantidade')*(veiculo.tempo_carga + veiculo.
tempo_descarga))
        if entrega.tempo_restante - tempo > 0:
            if (entrega.volume + Gr.node[v].get('volume') < veiculo.volume_max
and
                entrega.valor + Gr.node[v].get('valor') < veiculo.valor_max):
                return True
        return False
```

- **Classe Roteiro:**

  Armezana dados sobre as rotas obtidas para cada região.

In [39]:
```
class Roteiro:

    def __init__(self, G, regiao, r, veiculos, carga_horaria):
        # criar o subGrafo da regiao
        self.Gr = G.subgraph(regiao).copy()
        self.completa()
        self.centro = r

        #lista de entregas
        self.rotas = []

        #calcular menor caminho de todos para todos
        self.matriz = self.caminhoMinimo(self.Gr)

        #lista de veiculos
        self.veiculos = []
        self.quantVeiculos = []
        self.quantVeiculosUsados = []
        self.CustoPorKm = 0
        self.CustoPorHora = 0
        self.CustoFixo = 0
        self.CustoTotal = 0

        # dividir veiculos
        for tipo in veiculos.keys():
            self.veiculos.append(Veiculo(veiculos[tipo].volume_max, veiculos[ti
po].valor_max,
                                         veiculos[tipo].quantidade/5, veiculos[tipo].ve
locidade_centro,
                                         veiculos[tipo].velocidade, veiculos[tipo].temp
o_carga,
                                         veiculos[tipo].tempo_descarga, veiculos[tipo].
custo_hora,
                                         veiculos[tipo].custo_km, veiculos[tipo].custo_
fixo, tipo))
            self.quantVeiculos.append(veiculos[tipo].quantidade/5)
            self.quantVeiculosUsados.append(0)

        # chama a funcao roterizar
        self.roteirizar(r, carga_horaria)

    def completa(self):
        #completa o subgrafo Gr com as arestas entre os vertices representando
 a distancia
        for u in self.Gr.nodes():
            for v in self.Gr.nodes():
                if u != v:
                    distancia = calcula_distancia(self.Gr,u, v)
                    self.Gr.add_edge(u, v, distancia=distancia)

    def caminhoMinimo(self, Gr):
        #prenche o dicionario com a distancia direta entre os vertices - grafo
 kn
        distancias = {}
        for i in Gr:
            dicionario = {}
            for j in Gr:
                dicionario[j] = 0
            distancias[i] = dicionario
        for u in Gr:
            for v in Gr:
                distancia = ((Gr.nodes[u].get('x') - Gr.nodes[v].get('x'))**2 +
                             (Gr.nodes[u].get('y') - Gr.nodes[v].get('y'))**2)*
*(1/2)
                distancias[u][v] = distancia
        return distancias
```

```
    def roteirizar(self,r, carga_horaria):
        #a lista recebe todos os vertices contidos na regiao
        clientes_nao_atendidos = []
        for i in self.matriz:
            clientes_nao_atendidos.append(i)

        #enquanto nao atendemos todos os vertices na regiao, rodamos o loop
        #se sobrar apenas o centro, atendemos todos os vertices da regiao
        while clientes_nao_atendidos != [r]:
            #entregas
            entregas = []

            #cria uma entrega com a carga horaria maxima
            entrega = Entrega(carga_horaria)

            #encontra-se um veiculo disponivel para atender a rota
            veiculo = self.veiculoDisponivel()

            #adicionamos o veiculo a entrega
            entrega.veiculo = veiculo
            #adiciona centro
            u = r
            entrega.rota.append(u)

            #ordena os vizinhos de u de forma crescente por distancia
            Adj = self.listOrdCres(self.matriz[u], clientes_nao_atendidos, u, r
)
            #encontra o primeiro cliente
            encontrou = False
            for v in Adj:
                #verifica se v é um vertice seguro
                encontrou = entrega.prossegue(self.Gr, self.matriz, u, v, r, en
trega, veiculo)
                #se encontrou um vertice segura, adiciona-o na entrega e atuali
za o somatorios da entrega
                if encontrou:
                    entrega.rota.append(v)
                    entrega.km += self.matriz[u][v]
                    entrega.tempo_restante -= (self.matriz[u][v] / veiculo.velo
cidade_centro +
                                               self.Gr.nodes[v].get('quantidade'
) *
                                               (veiculo.tempo_carga + veiculo.te
mpo_descarga))
                    entrega.volume += self.Gr.nodes[v].get('volume')
                    entrega.valor += self.Gr.nodes[v].get('valor')
                    u = v
                    break

            #percorre a regiao para adicionar os clientes intermediarios na rot
a,
            nao_encontrou = False
            while not nao_encontrou:
                #booleano para verificar se no loop abaixo encontrou-se algum c
liente para adicionar na rota
                nao_encontrou = True
                encontrou = False
                Adj = self.listOrdCres(self.matriz[u],clientes_nao_atendidos, u
, r)
                for v in Adj:
                    if v not in entrega.rota:
                        encontrou = entrega.prossegue(self.Gr, self.matriz, u,
v, r, entrega, veiculo)
                        if encontrou:
                            entrega.rota.append(v)
                            entrega.km += self.matriz[u][v]
                            entrega.tempo_restante -= (self.matriz[u][v] / veic
ulo.velocidade_centro +
```

```
                                                        self.Gr.nodes[v].get('quantidade') *
                                                        (veiculo.tempo_carga + veiculo.tempo_descarga))
                            entrega.volume += self.Gr.nodes[v].get('volume')
                            entrega.valor += self.Gr.nodes[v].get('valor')
                            u = v
                            nao_encontrou = False
                            break
            #retorno para o centro
            entrega.rota.append(r)
            entrega.km += self.matriz[r][u]
            entrega.tempo_restante -= self.matriz[r][u] / veiculo.velocidade_centro
            self.rotas.append(entrega)
            clientes_nao_atendidos.append(r)
        #calcular o somatorio de custos da regiao
        self.calcularCustos()


    def veiculoDisponivel(self):
        # retorna um veiculo disponivel e atualiza os valores referentes a ele
        indice = 0
        for v in self.veiculos:
            if self.quantVeiculos[indice] > 0:
                self.quantVeiculos[indice] -= 1
                self.quantVeiculosUsados[indice] +=1
                return v
            indice += 1

    def listOrdCres(self, matriz, keys, u, r):
        # ordena a lista de chaves pela distancia para o vertice u
        if u in keys:
            keys.remove(u)
        if u != r and r in keys:
            keys.remove(r)

        listOrdCres = []
        #adiciona as chaves em outra lista
        for i in keys:
            listOrdCres.append(i)
        n = len(keys)
        #ordena a lista de saida pela distancia
        for i in range(0, n):
            for j in range(i+1, n):
                if matriz[listOrdCres[i]] > matriz[listOrdCres[j]]:
                    aux = listOrdCres[i]
                    listOrdCres[i] = listOrdCres[j]
                    listOrdCres[j] = aux
        return listOrdCres

    def calcularCustos(self):
        #custo fixo, custo por hora e por km
        for entrega in self.rotas:
            self.CustoFixo += entrega.veiculo.custo_fixo
            self.CustoPorHora += entrega.veiculo.custo_hora * (7 - entrega.tempo_restante)
            self.CustoPorKm += entrega.veiculo.custo_km * entrega.km
            self.CustoTotal += self.CustoFixo + self.CustoPorHora + self.CustoPorKm


    def imprimir(self):
        # imprime os dados das rotas
        print('Rotas da regiao ' + str(self.centro))
        for rota in self.rotas:
            print('rota:', rota.rota, 'veiculo: ', rota.veiculo.tipo)
        print('Custo por KM = ', self.CustoPorKm)
```

```
        print('Custo por Hora = ', self.CustoPorHora)
        print('Custo Fixo = ', self.CustoFixo)
        print('Somatório de Custos = ', self.CustoPorKm + self.CustoPorHora + s
elf.CustoFixo)
```

Função completa o grafo da região, adicionando arestas entre todos os vértices, sendo cada aresta a distancia entre os vértices.

In [14]:
```
def completa(self):
    #completa o subgrafo Gr com as arestas entre os vertices representando a di
stancia
    for u in self.Gr.nodes():
        for v in self.Gr.nodes():
            if u != v:
                distancia = calcula_distancia(self.Gr,u, v)
                self.Gr.add_edge(u, v, distancia=distancia)
```

Função caminho mínimo é resposável por criar a matriz de caminhos minimos de todos os vertices

In [15]:
```
def caminhoMinimo(self, Gr):
    #prenche o dicionario com a distancia direta entre os vertices - grafo kn
    distancias = {}
    for i in Gr:
        dicionario = {}
        for j in Gr:
            dicionario[j] = 0
        distancias[i] = dicionario
    for u in Gr:
        for v in Gr:
            distancia = ((Gr.nodes[u].get('x') - Gr.nodes[v].get('x'))**2 +
                         (Gr.nodes[u].get('y') - Gr.nodes[v].get('y'))**2)**(1/
2)
            distancias[u][v] = distancia
    return distancias
```

Principal função da classe roteiro: Nela são criadas as rotas da região, definindo os clientes pertencentes a uma rota(classe entrega) e o veiculo (classe veiculo).

In [16]:
```
def roteirizar(self,r, carga_horaria):
    #a lista recebe todos os vertices contidos na regiao
    clientes_nao_atendidos = []
    for i in self.matriz:
        clientes_nao_atendidos.append(i)
    #enquanto nao atendemos todos os vertices na regiao, rodamos o loop
    #se sobrar apenas o centro, atendemos todos os vertices da regiao
    while clientes_nao_atendidos != [r]:
        #entregas
        entregas = []

        #cria uma entrega com a carga horaria maxima
        entrega = Entrega(carga_horaria)

        #encontra-se um veiculo disponivel para atender a rota
        veiculo = self.veiculoDisponivel()

        #adicionamos o veiculo a entrega
        entrega.veiculo = veiculo
        #adiciona centro
        u = r
        entrega.rota.append(u)

        #ordena os vizinhos de u de forma crescente por distancia
        Adj = self.listOrdCres(self.matriz[u], clientes_nao_atendidos, u, r)
        #encontra o primeiro cliente
        encontrou = False
        for v in Adj:
            #verifica se v é um vertice seguro
            encontrou = entrega.prossegue(self.Gr, self.matriz, u, v, r, entreg
a, veiculo)
            #se encontrou um vertice segura, adiciona-o na entrega e atualiza o
 somatorios da entrega
            if encontrou:
                entrega.rota.append(v)
                entrega.km += self.matriz[u][v]
                entrega.tempo_restante -= self.matriz[u][v] / veiculo.velocidad
e_centro + self.Gr.nodes[v].get('quantidade') * (veiculo.tempo_carga + veiculo.
tempo_descarga)
                entrega.volume += self.Gr.nodes[v].get('volume')
                entrega.valor += self.Gr.nodes[v].get('valor')
                u = v
                break

        #percorre a regiao para adicionar os clientes intermediarios na rota,
        nao_encontrou = False
        while not nao_encontrou:
            #booleano para verificar se no loop abaixo encontrou-se algum clien
te para adicionar na rota
            nao_encontrou = True
            encontrou = False
            Adj = self.listOrdCres(self.matriz[u],clientes_nao_atendidos, u, r)
            for v in Adj:
                if v not in entrega.rota:
                    encontrou = entrega.prossegue(self.Gr, self.matriz, u, v, r
, entrega, veiculo)
                    if encontrou:
                        entrega.rota.append(v)
                        entrega.km += self.matriz[u][v]
                        entrega.tempo_restante -= self.matriz[u][v] / veiculo.v
elocidade_centro + self.Gr.nodes[v].get('quantidade') * (veiculo.tempo_carga +
veiculo.tempo_descarga)
                        entrega.volume += self.Gr.nodes[v].get('volume')
                        entrega.valor += self.Gr.nodes[v].get('valor')
                        u = v
                        nao_encontrou = False
                        break
```

```
        #retorno para o centro
        entrega.rota.append(r)
        entrega.km += self.matriz[r][u]
        entrega.tempo_restante -= self.matriz[r][u] / veiculo.velocidade_centro
        self.rotas.append(entrega)
        clientes_nao_atendidos.append(r)
    #calcular o somatorio de custos da regiao
    self.calcularCustos()
```

Abaixo outras funções da classe roteiro: A primeira, a função veículo disponível, retorna um veículo disponível na região.

In [17]:
```
#retorna um veiculo disponivel e atualiza os valores referentes a ele
def veiculoDisponivel(self):
    indice = 0
    for v in self.veiculos:
        if self.quantVeiculos[indice] > 0:
            self.quantVeiculos[indice] -= 1
            self.quantVeiculosUsados[indice] +=1
            return v
        indice += 1
```

A função listOrdCres() é responsável por retornar os vértices, que ainda estão sem rota especificada, em ordem crescente de distancia a partir do vértice u, que é o último vértice a ser inserido na rota.

In [18]:
```
#ordena a lista de chaves pela distancia para o vertice u
def listOrdCres(self, matriz, keys, u, r):
    if u in keys:
        keys.remove(u)
    if u != r and r in keys:
        keys.remove(r)

    listOrdCres = []
    #adiciona as chaves em outra lista
    for i in keys:
        listOrdCres.append(i)
    n = len(keys)
    #ordena a lista de saida pela distancia
    for i in range(0, n):
        for j in range(i+1, n):
            if matriz[listOrdCres[i]] > matriz[listOrdCres[j]]:
                aux = listOrdCres[i]
                listOrdCres[i] = listOrdCres[j]
                listOrdCres[j] = aux
    return listOrdCres
```

Função que calcula os custos da região. Somatório do custo fixo de cada veiculo, do custo por hora de cada veiculo e do custo por km de cada, além do custo total.

In [19]:
```
def calcularCustos(self):
    #custo fixo, custo por hora e por km
    for entrega in self.rotas:
        self.CustoFixo += entrega.veiculo.custo_fixo
        self.CustoPorHora += entrega.veiculo.custo_hora * (7 - entrega.tempo_re
stante)
        self.CustoPorKm += entrega.veiculo.custo_km * entrega.km
        self.CustoTotal += self.CustoFixo + self.CustoPorHora + self.CustoPorKm
```

Função auxiliar para imprimir os custos calculados

In [20]:
```
#imprime os dados das rotas
def imprimir(self):
    print('Rotas da regiao ' + str(self.centro))
    for rota in self.rotas:
        print('rota:', rota.rota, 'veiculo: ', rota.veiculo.tipo)
    print('Custo por KM = ', self.CustoPorKm)
    print('Custo por Hora = ', self.CustoPorHora)
    print('Custo Fixo = ', self.CustoFixo)
    print('Somatório de Custos = ', self.CustoPorKm + self.CustoPorHora + self.
CustoFixo)
```

In [41]:
```
# 2a etapa: cria rotas para as regiões
roteiros = []

#criar as rotas
for r, regiao in enumerate(regioes):
    roteiros.append(Roteiro(G, regiao, r, veiculos, carga_horaria))

#imprimir os resultados
for rota in roteiros:
    rota.imprimir()
    print()
```

```
Rotas da regiao 0
rota: [0, 34, 32, 71, 54, 63, 11, 65, 8, 0] veiculo:  van
rota: [0, 56, 45, 73, 81, 82, 84, 96, 0] veiculo:  van
rota: [0, 68, 95, 0] veiculo:  van
Custo por KM =  507.0152234063621
Custo por Hora =  864.5592319929531
Custo Fixo =  516
Somatório de Custos =  1887.5744553993152

Rotas da regiao 1
rota: [1, 35, 67, 62, 70, 9, 7, 1] veiculo:  van
rota: [1, 74, 40, 17, 14, 1] veiculo:  van
rota: [1, 23, 88, 57, 1] veiculo:  van
rota: [1, 26, 1] veiculo:  minivan
rota: [1, 25, 1] veiculo:  minivan
rota: [1, 47, 1] veiculo:  minivan
rota: [1, 46, 49, 90, 1] veiculo:  carro
rota: [1, 38, 1] veiculo:  carro
Custo por KM =  1310.8462583421729
Custo por Hora =  2265.6687329344145
Custo Fixo =  1153
Somatório de Custos =  4729.514991276587

Rotas da regiao 2
rota: [2, 24, 12, 30, 22, 36, 91, 2] veiculo:  van
rota: [2, 86, 37, 15, 83, 94, 2] veiculo:  van
rota: [2, 64, 80, 61, 2] veiculo:  van
rota: [2, 76, 43, 78, 2] veiculo:  minivan
rota: [2, 39, 89, 59, 2] veiculo:  minivan
rota: [2, 85, 6, 2] veiculo:  minivan
rota: [2, 42, 27, 2] veiculo:  carro
rota: [2, 53, 2] veiculo:  carro
Custo por KM =  1212.8472265052249
Custo por Hora =  2396.1895434054068
Custo Fixo =  1153
Somatório de Custos =  4762.036769910632

Rotas da regiao 3
rota: [3, 77, 99, 41, 21, 10, 51, 3] veiculo:  van
rota: [3, 72, 98, 97, 69, 55, 3] veiculo:  van
rota: [3, 31, 44, 19, 13, 3] veiculo:  van
rota: [3, 92, 3] veiculo:  minivan
Custo por KM =  565.1484273652786
Custo por Hora =  1154.7986153043523
Custo Fixo =  669
Somatório de Custos =  2388.9470426696307

Rotas da regiao 4
rota: [4, 28, 18, 20, 79, 33, 75, 66, 4] veiculo:  van
rota: [4, 16, 93, 58, 4] veiculo:  van
rota: [4, 48, 29, 60, 87, 52, 5, 4] veiculo:  van
rota: [4, 50, 4] veiculo:  minivan
Custo por KM =  942.2933886557278
Custo por Hora =  1236.2085528111547
Custo Fixo =  669
Somatório de Custos =  2847.5019414668823
```