```python
#!/usr/bin/env python
# -*- coding: utf8 -*-

"""
   Todo lo relacionado al uso de grafo y el grafo mismo
"""


from constantes import *
from texto import *


class Grafo:
    def __init__(self):
        """Inicializa el grafo"""
        self.vertices = {}

    def __iter__(self):
        """ Devuelve un iterador de los vertices"""
        return iter(self.vertices.values())

    def __contains__(self, vertice):
        """si el vertice esta en el grafo, debuelve True"""
        if vertice in self.vertices:
            return True
        return False

    def __len__(self):
        "Cantidad de vertices"
        return len(self.vertices)

    def agregar_vertice(self, clave):
        """Agrega un vertice"""
        self.vertices[clave] = Vertice(clave)

    def obtener_vertice(self, vertice):
        """Devuelve un objeto vertice"""
        if vertice in self.vertices:
            return self.vertices[vertice]
        else:
            return None

    def obtener_lista_vertices(self):
        """Devuelve todos los vertices del grafo (str)"""
        return self.vertices.keys()

    def obtener_objetos_vertice(self):
        """Devuelve todos los vertieces (OBJETOS)"""
        return self.vertices.values()


    def agregar_arista(self, vertice, vecino, peso):
        """Agrega un vertice adyacente al actual con el peso"""
        if vertice not in self.vertices:
            self.agregar_vertice(vertice)
        if vecino not in self.vertices:
            self.agregar_vertice(vecino)
        self.vertices[vertice].agregar_vecino(self.vertices[vecino], peso)

class Vertice:
    def __init__(self, clave):
        """Inicializa el vertice"""
        self.adyacentes = {}
        self.clave = clave

```

```
66      def agregar_vecino(self, vecino, peso):
67          """Agrega vertices adyacentes al vertice"""
68          self.adyacentes[vecino] = peso
69
70      def obtener_adyacentes(self):
71          """Devuelve sus vertices adyacentes"""
72          return self.adyacentes.keys()
73
74      def obtener_lista_adyacentes(self):
75          return [int(i.clave) for i in self.adyacentes]
76
77      def obtener_clave(self):
78          """Devuelve su 'nombre'"""
79          return self.clave
80
81      def obtener_peso(self, vecino):
82          """Devuelve el peso de la arista"""
83          return self.adyacentes[vecino]
84
85
86  def validar_calles(calles, grafo):
87      """Verifica que las calles recibidas (dupla) esten dendo del grafo"""
88      verticeA = grafo.obtener_vertice(calles[0])
89      verticeB = grafo.obtener_vertice(calles[1])
90
91      if verticeA and verticeB:
92          #reviso que sea una haya interseccion
93          for vertice in verticeA.obtener_adyacentes():
94              if vertice in verticeB.obtener_adyacentes():
95                  return vertice
96          #reviso desde el otro lado
97          for vertice in verticeB.obtener_adyacentes():
98              if vertice in verticeA.obtener_adyacentes():
99                  return vertice
100     return False
101
102 def imprimir_distancia(distancia):
103     """Recibe un diccionario e imprime las distancias de manera
104     leible para los.. humanos"""
105     salida = []
106
107     for vertice, distancia in distancia.items():
108         if distancia == infinito:
109             distancia = "INF"
110         cad = str(vertice.clave) + " - " +  str(distancia)
111         salida.append(cad)
112     salida.sort()
113     for elemento in salida:
114         print elemento
115
116
117 def procesar_ruta(ruta, vert_inicio, vert_fin):
118     """Recibe una ruta (diccionario) el vertice inicial y el vertice
119     final, devuelve una lista de los vertices(objeto) ordenados de
120     de inicio a fin"""
121
122     salida = []
123     cadena =  ""
124     aux = ruta[vert_fin]
125     salida.append(vert_fin)
126
127     try:
128         for i in range(len(ruta)-2):
129             salida.append(aux)
130             aux = ruta[aux]
```

```
131         salida.append(aux)
132     except KeyError:
133         #Es por el primer/ultimo vertice
134         pass
135 
136     salida.append(vert_inicio)
137 
138     salida.reverse()
139 
140     return salida
141 
142 
143 def imprimir_ruta(ruta, vert_inicio, vert_fin):
144     """Imprime la ruta de manera legible"""
145     salida = []
146     i = 0
147     cadena =  ""
148 
149     aux = ruta[vert_fin]
150     salida.append(str(vert_fin.clave))
151 
152     try:
153         for i in range(len(ruta)-2):
154 
155             salida.append(str(aux.clave))
156             aux = ruta[aux]
157         salida.append(str(aux.clave))
158 
159     except KeyError:
160         #Es por el primer/ultimo vertice
161         pass
162     salida.append(str(vert_inicio.clave))
163     salida.reverse()
164     for item in salida:
165         cadena += item+"->"
166 
167     print cadena[:len(cadena)-2]
168 
169 
170 
171 
172 
173 def dijkstra(grafo,nodo_inicial):
174     """Algoritmo Dijkstra, genera los mejores caminos desde el nodo_inicial
175     al resto de los nodos"""
176     #basado en el de wikipedia y en los de:
177     #https://code.activestate.com/recipes/119466-dijkstras-algorithm-for-sh
    ortest-paths/
178 
179     ruta = {}
180     distancia = {}
181     visto = {}
182 
183     for vertice in grafo.obtener_objetos_vertice():
184         #Marco todos los vertices en distancia infinita y como no vistos
185         distancia[vertice] = infinito
186         visto[vertice] = False
187         if vertice in nodo_inicial.obtener_adyacentes():
188             #Si son adyacentes al nodo_inicial, le cargo sus pesos
189             distancia[vertice] = float(nodo_inicial.obtener_peso(vertice))
190 
191     #Remarco el nodo inicial a 0 y como visitado
192     distancia[nodo_inicial] = 0
193     visto[nodo_inicial]=True
194 
```

```
195     while False in visto.values():
196         #mientras haya nodos no vistos, busco el vertice adyacente
197         #mas liviano (digamos...)
198         vertice=_buscar_adyacente_minimo(distancia,visto)
199         visto[vertice] = True
200         for elemento in vertice.obtener_adyacentes():
201             #Busco el menor y lo agrego a la ruta
202             if distancia[elemento] > distancia[vertice] + float(vertice.obt
    ener_peso(elemento)):
203                 distancia[elemento] = distancia[vertice] + float(vertice.ob
    tener_peso(elemento))
204                 ruta[elemento] = vertice
205 
206     return ruta
207 
208 def _buscar_adyacente_minimo(distancia,visitados):
209     """Funcion PRIVADA, Busca el vertice mas cercano y menos pesado"""
210     minimo_peso = infinito
211     vertice = object
212 
213     #reviso cada elemento, con su respectivo estado
214     for elemento, estado in visitados.items():
215         #Busco el vertice mas cercano no visitado
216         if distancia[elemento] <= minimo_peso and estado == False:
217             vertice = elemento
218             minimo_peso = distancia[elemento]
219     return vertice
220 
221 def parsear_ruta(ruta, info_nodos):
222     """Recibe un ruta con el sigueinte estilo:
223     [vert1,vart2,...] y la devuelve de la menera x,y x,y x,y"""
224     salida = []
225     for vertice in ruta:
226         lat = str(info_nodos[vertice.clave]["lat"])
227         lon = str(info_nodos[vertice.clave]["lon"])
228         salida.append(lat+","+lon)
229     return salida
230 
231 def viaje(grafo_nodos, A, nombreA, B, nombreB, info_nodos, kml):
232     """Genera el recorrido y lo guarda en el kml"""
233     verticeA = A[0]
234     dir_verticeA = A[1]
235     info_verticeA = A[2]
236 
237     verticeB = B[0]
238     dir_verticeB = B[1]
239     info_verticeB = B[2]
240 
241     print msj_dijkstra
242     ruta = dijkstra(grafo_nodos, verticeA)
243 
244     ruta_procesada = procesar_ruta(ruta, verticeA, verticeB)
245     ruta_parseada = parsear_ruta(ruta_procesada, info_nodos)
246     #~ Genero el kml
247     print msj_kml
248     info_verticeA = info_nodos[verticeA.clave]
249     info_verticeB = info_nodos[verticeB.clave]
250     kml.agregar_marcador(texto(nombreA)+" "+dir_verticeA[0]+" y "+dir_verti
    ceA[1], info_verticeA["lat"], info_verticeA["lon"])
251     kml.agregar_marcador(texto(nombreB)+" "+dir_verticeB[0]+" y "+dir_verti
    ceB[1], info_verticeB["lat"], info_verticeB["lon"])
252     kml.agregar_ruta("Ruta "+texto(nombreA)+"->"+texto(nombreB), ruta_parse
    ada)
253 
254 
```

255
256
257