Reporte de agente viajero (algoritmo de Kruskal)

Problema del agente viajero:

El problema del agente viajero (PAV, por sus siglas) responde a la siguiente pregunta "Dada una lista de ciudades y la distancia entre cada par de ellas ¿Cuál es la ruta más corta posible para que visite cada ciudad solo una vez y al final regresar a la ciudad de origen?".

Traduciendo este problema en grafos: dado un grafo completo, determinar el ciclo hamiltoneano más corto, donde este ciclo se refiere a un camino del grafo visitando todos los vértices de este y que el ultimo vértice visitado es el adyacente al primero. El peso del camino es la suma de las aristas que lo componen.

Descripción del algoritmo de Kruskal

Para implementar Kruskal necesitamos un grafo árbol (un árbol es un grafo conexo con n vértices y n-1 aristas, al ser conexo se dice que todos los vértices del grafo están conectados, pero al ser un árbol, este camino es único).

Este algoritmo primero ordena todas las aristas de menor a mayor peso, después intenta unir cada arista cuidando que no se haga un ciclo.

Código de algoritmo de Kruskal:

```
from copy import deepcopy
import random

def permutaciones(arr):
    if len(arr)==0: return [[]]
    perm=[]
    for i in range(len(arr)):
        wot=permutaciones(arr[:i]+arr[i+1:])
        for w in wot:
            perm.append([arr[i]]+w)
    return perm

class grafo:
    def __init__(self):
        self.V =set() #un conjunto
        self.E = dict() #un mapeo de pesos a aritstas
        self.vecinos = dict() #un mapeo

    def agrega(self, v ):
        self.V.add(v)
        if not  v in self.vecinos: # vecindad de v
            self.vecinos[v]= set() #inicialmente no tiene nada
    def conecta(self, v , u , peso = 1):
        self.agrega(v)
        self.agrega(u)
        self.E[(v,u)] = self.E[(u,v)] = peso
        self.vecinos[v].add(u)
        self.vecinos[u].add(v)

    def complemento(self):
        comp= grafo()
        for v in self.V:

           for w in self.V:
               if v != w  and (v,w) not in self.E:
                   comp.conecta(v,w,1)
        return comp

    def aristas(self):
        return self.E

    def vertices(self):
        return self.V

    def __str__(self):
        return "Aristas= " + str(self.E)+"\nVertices = " +str(self.V)

    def BFS_n(self, bi):
        visitados=[]          ##arreglo con nodos visitados inicialmente vacio
        Xvisitar=fila()      ##fila con los nodos por visitar
        Xvisitar.meter( bi )
        while Xvisitar.longitud > 0: ##mientras haya alguien en fila
            nodo = Xvisitar.obtener()
            if nodo not in visitados:  ##si el nodo aun no en visitado
                visitados.append(nodo)
                for vecino in self.vecinos[nodo]:
                    Xvisitar.meter(vecino)
        return visitados

    def DFS_n(self, bi):
        visitados=[]          ##arreglo con nodos visitados inicialmente vacio
        Xvisitar=pila()      ##fila con los nodos por visitar
        Xvisitar.meter( bi )
        while Xvisitar.longitud > 0: ##mientras haya alguien en fila
            nodo = Xvisitar.obtener()
            if nodo not in visitados:  ##si el nodo aun no en visitado
```

```
            visitados.append(nodo)
            for vecino in self.vecinos[nodo]:
                Xvisitar.meter(vecino)
    return visitados

def DFS_n(self, bi):
    visitados=[]          ##arreglo con nodos visitados inicialmente vacio
    Xvisitar=pila()      ##fila con los nodos por visitar
    Xvisitar.meter( bi )
    while Xvisitar.longitud > 0: ##mientras haya alguien en fila
        nodo = Xvisitar.obtener()
        if nodo not in visitados:  ##si el nodo aun no en visitado
            visitados.append(nodo)
            for vecino in self.vecinos[nodo]:
                Xvisitar.meter(vecino)
    return visitados

def kruskal(self):
    e = deepcopy(self.E)
    arbol = grafo()
    peso = 0
    comp = dict()
    t = sorted(e.keys(), key = lambda k: e[k], reverse=True)
    nuevo = set()
    while len(t) > 0 and len(nuevo) < len(self.V):
        #print(len(t))
        arista = t.pop()
        w = e[arista]
        del e[arista]
        (u,v) = arista
        c = comp.get(v, {v})
        if u not in c:
```

```
            #print('u ',u, 'v ',v ,'c ', c)
            arbol.conecta(u,v,w)
            peso += w
            nuevo = c.union(comp.get(u,{u}))
            for i in nuevo:
                comp[i]= nuevo
    print('MST con peso', peso, ':', nuevo, '\n', arbol.E)
    return arbol


def DFS(self,bi):
    visitados =[]
    f=pila()
    f.meter(bi)
    while(f.longitud>0):
        na = f.obtener()
        visitados.append(na)
        ln = self.vecinos[na]
        for nodo in ln:
            if nodo not in visitados:
                f.meter(nodo)
    return visitados

def vecinoMasCercano(self):
    lv = list(self.V)   ##lista de vertices
    random.shuffle(lv)
    ni = lv.pop()
    inicial = ni
    lv2=list()
    lv2.append(ni)
    peso=0
    while len(lv2)<len(self.V):
```

```
        le = list()
        ln=list()
        ln = self.vecinos[ni]
        for nv in ln:
            if nv not in lv2:
                le.append((nv,self.E[(ni,nv)]))
        sorted(le, key = lambda le: le[1] )
        t=le[0]
        lv2.append(t[0])
        peso=peso+t[1]
        ni=t[0]
    peso=peso+self.E[lv2[-1], inicial]
    lv2.append(inicial)
    return (lv2,peso)


def Problem(self):
    perm=permutaciones(list(self.V))
    mejor=-1
    camino=[]
    for w in perm:
        peso=0
        for i in range(len(w)-1):
            peso+=self.E[(w[i],w[i+1])]
        peso+=self.E[(w[-1],w[0])]
        if peso<mejor or mejor==-1:
            mejor=peso
            camino=w
    return camino
```

```
    visitados= dict()    ##dicionario con llaves igual a nodos y valores igual a distancia de nodo inicial
    Xvisitar=fila()
    Xvisitar.meter(  (ni,0)   )
    while Xvisitar.longitud > 0: ##mientras haya alguien en fila
        nodo = Xvisitar.obtener()
        if nodo[0] not in visitados:
            visitados[nodo[0]]=nodo[1]
            for vecino in g.vecinos[nodo[0]]:
                #vecinos_d.append( (e,nodo[1]+1) )
                Xvisitar.meter((vecino,nodo[1]+1))
            #for v in vecinos_d:
                #f.meter(v)
    return visitados

def DFS_N(g, bi):
    visitados= dict()    ##dicionario con llaves igual a nodos y valores igual a distancia de nodo inicial
    Xvisitar=pila()
    Xvisitar.meter(  (bi,0)   )
    while Xvisitar.longitud > 0: ##mientras haya alguien en fila
        nodo = Xvisitar.obtener()
        if nodo[0] not in visitados:
            visitados[nodo[0]]=nodo[1]
            for vecino in g.vecinos[nodo[0]]:
                #vecinos_d.append( (e,nodo[1]+1) )
                Xvisitar.meter((vecino,nodo[1]+1))
            #for v in vecinos_d:
                #f.meter(v)
    return visitados



class pila(object):##quitas el mas nuevo STACK
    def __init__(self):
        self.a=[]

    def obtener(self):
        return self.a.pop()

    def meter(self, e):
        self.a.append(e)

    @property
    def longitud(self):
        return len(self.a)

    def __str__(self):
        return "<" + str(self.a)+ ">"




class fila(pila):##quitas el que ha estado mas tiempo
    def obtener(self):
        return self.a.pop(0)
```

```
g=grafo()
            g.conecta("Sarahi","Ismael",0.55)
            g.conecta("Sarahi","Diego",0.48)
            g.conecta("Sarahi ","Tapia",6.4)
            g.conecta("Sarahi ","Asael",2.24)
            g.conecta("Sarahi ","Rodrigo",6.26)
            g.conecta("Sarahi ","Memo",1.96)
            g.conecta("Sarahi ","Lozoya",6.4)
            g.conecta("Sarahi ","Thamara",2.24)
            g.conecta("Sarahi ","Pilar",6.26)


            g.conecta("Ismael","Sarahi",0.55)
            g.conecta("Ismael","Diego",0.38)
            g.conecta("Ismael","Tapia",6.56)
            g.conecta("Ismael","Asael",1.64)
            g.conecta("Ismael","Rodrigo",6.36)
            g.conecta("Ismael","Memo",2.49)
            g.conecta("Ismael","Lozoya",0.78)
            g.conecta("Ismael","Thamara",2.79)
            g.conecta("Ismael","Pilar",3.76)


            g.conecta("Diego","Sarahi",0.48)
            g.conecta("Diego"," Ismael",0.38)
            g.conecta("Diego","Tapia",6.27)
            g.conecta("Diego","Asael",2.02)
            g.conecta("Diego","Rodrigo",6.62)
            g.conecta("Diego","Memo",2.42)
            g.conecta("Diego","Lozoya",1.22)
            g.conecta("Diego","Thamara",2.57)
            g.conecta("Diego","Pilar",4.14)


            g.conecta("Tapia", "Sarahi",6.4)
            g.conecta("Tapia", " Ismael",6.56)
            g.conecta("Tapia", "Diego",6.27)
            g.conecta("Tapia", "Asael",8.3)
            g.conecta("Tapia", "Rodrigo",9.38)
            g.conecta("Tapia", "Memo",4.61)
            g.conecta("Tapia", "Lozoya",7.46)
            g.conecta("Tapia", "Thamara",4.15)
            g.conecta("Tapia", "Pilar",10.4)
```

```
g.conecta("Asael","Sarahi",2.24)
g.conecta("Asael"," Ismael",1.64)
g.conecta("Asael","Diego",2.02)
g.conecta("Asael","Tapia",8.30)
g.conecta("Asael","Rodrigo",6.45)
g.conecta("Asael","Memo",4.03)
g.conecta("Asael","Lozoya",0.87)
g.conecta("Asael","Thamara",4.38)
g.conecta("Asael","Pilar",2.12)


g.conecta("Rodrigo","Sarahi",6.26)
g.conecta("Rodrigo"," Ismael",6.36)
g.conecta("Rodrigo","Diego",6.62)
g.conecta("Rodrigo","Tapia",9.38)
g.conecta("Rodrigo","Asael",6.45)
g.conecta("Rodrigo","Memo",5.61)
g.conecta("Rodrigo","Lozoya",6.11)
g.conecta("Rodrigo","Thamara",5.98)
g.conecta("Rodrigo","Pilar",6.9)


g.conecta("Memo","Sarahi",1.96)
g.conecta("Memo"," Ismael",2.49)
g.conecta("Memo","Diego",2.42)
g.conecta("Memo","Tapia",4.61)
g.conecta("Memo","Asael",4.03)
g.conecta("Memo","Rodrigo",5.61)
g.conecta("Memo","Lozoya",3.17)
g.conecta("Memo","Thamara",0.47)
g.conecta("Memo","Pilar",5.95)


g.conecta("Lozoya","Sarahi",1.41)
g.conecta("Lozoya","Imael",0.78)
g.conecta("Lozoya","Diego", 122)
g.conecta("Lozoya","Tapia",7.46)
g.conecta("Lozoya","Asael",0.87)
g.conecta("Lozoya","Rodrigo",6.11)
g.conecta("Lozoya","Memo",3.17)
g.conecta("Lozoya","Thamara",3.53)
```

```
g.conecta("Lozoya","Pilar",2.9)


g.conecta("Thamara","Sarahi",2.22)
g.conecta("Thamara","Ismael",2.79)
g.conecta("Thamara","Diego",2.57)
g.conecta("Thamara","Tapia",4.15)
g.conecta("Thamara","Asael",4.38)
g.conecta("Thamara","Rodrigo",5.98)
g.conecta("Thamara","Memo",0.43)
g.conecta("Thamara","Lozoya",3.53)
g.conecta("Thamara","Pilar",6.35)


g.conecta("Pilar"," Sarahi ",4.33)
g.conecta("Pilae","Ismael",3.76)
g.conecta("Pilar","Diego",4.14)
g.conecta("Pilar","Tapia",10.4)
g.conecta("Pilar","Asael",2.12)
g.conecta("Pilar","Rodrigo",6.9)
g.conecta("Pilar","Memo ",5.95)
g.conecta("Pilar","Lozoya",2.9)
g.conecta("Pilar","Thamara",6.35)


#EMPLEANDO KRUSKAL EN EL ALGORITMO DE APROXIMACION
C=g.kruskal()
for i in range(50):
    bi=random.choice(list(C.V))
    dfs=C.DFS(bi)
    c=0
    for f in range(len(dfs)-1):
        c+=g.E[(dfs[f],dfs[f+1])]
        print(dfs[f],dfs[f+1],g.E[(dfs[f],dfs[f+1])])
    c+=g.E[(dfs[-1],dfs[0])]
    print(dfs[-1],dfs[0],g.E[(dfs[-1],dfs[0])])
    print("costo:",c,"\n")
```

Algoritmo de Dijkstra:

Este algoritmo nos dice cuál es el camino más corto dado un nodo inicial hacia los demás nodos en un grafo con peso en cada arista

Código de algoritmo de Dijkstra:

```python
from heapq import heappop, heappush

def flatten(L):
    while len(L) > 0:
        yield L[0]
        L = L[1]

class Grafo:

    def __init__(self):
        self.V = set()
        self.E = dict()
        self.vecinos = dict()

    def agrega(self, v):
        self.V.add(v)
        if not v in self.vecinos:
            self.vecinos[v] = set()

    def conecta(self, v, u, peso=1):
        self.agrega(v)
        self.agrega(u)
        self.E[(v, u)] = self.E[(u, v)] = peso
        self.vecinos[v].add(u)
        self.vecinos[u].add(v)

    def complemento(self):
        comp= Grafo()
        for v in self.V:
            for w in self.V:
                if v != w and (v, w) not in self.E:
                    comp.conecta(v, w, 1)
        return comp

    def shortest(self, v):
        q = [(0, v, ())]
        dist = dict()
        visited = set()
        while len(q) > 0:
            (l, u, p) = heappop(q)
            if u not in visited:
                visited.add(u)
                dist[u] = (l,u,list(flatten(p))[::-1] + [u])
            p = (u, p)
            for n in self.vecinos[u]:
                if n not in visited:
                    el = self.E[(u,n)]
                    heappush(q, (l + el, n, p))
        return dist
```

#10 nodos, 20 aristas

```
f=Grafo()
f.conecta('Sarahi','Ismael',3)
f.conecta('Sarahi','Diego',9)
f.conecta('Sarahi','Tapia',12)
f.conecta('Sarahi','Lozoya',7)
f.conecta('Sarahi','Asael',4)
f.conecta('Sarahi','Rodrigo',3)
f.conecta('Sarahi','Thamara',2)
f.conecta('Sarahi','Pilar',1)
f.conecta('Sarahi','Memo',5)
f.conecta('Ismael','Diego',11)
f.conecta('Ismael','Tapia',8)
f.conecta('Diego','Tapia',9)
f.conecta('Tapia','Lozoya',4)
f.conecta('Lozoya','Asael',7)
f.conecta('Asael','Rodrigo',6)
f.conecta('Rodrigo','Thamara',1)
f.conecta('Thamara','Lozoya',10)
f.conecta('Pilar','Memo',14)
f.conecta('Memo','Thamara',4)
f.conecta('Memo','Rodrigo',5)
f.conecta('Diego','Lozoya',5)



print(f.shortest('Sarahi'))
```

Resultados:

```
#{'Sarahi': (0, 'Sarahi', ['Sarahi']), 'Pilar': (1, 'Pilar', ['Sarahi',
'Pilar']), 'Thamara': (2, 'Thamara', ['Sarahi', 'Thamara']), 'Ismael': (3,
'Ismael', ['Sarahi', 'Ismael']), 'Rodrigo': (3, 'Rodrigo', ['Sarahi',
'Rodrigo']),
#'f': (4, 'Asael', ['Sarahi', 'f']), 'Memo': (5, 'Memo', ['Sarahi', 'Memo']),
'Lozoya': (7, 'Lozoya', ['Sarahi', 'Lozoya']), 'Diego': (9, 'Diego',
['Sarahi', 'Diego']), 'Tapia': (11, 'Tapia', ['Sarahi', 'Ismael', 'Tapia'])}
```

Sarahi Sanchez Cruz

Matricula: 1743132

22 de mayo de 2018