**Villanueva Andrade Daniela Estefanía**

**1684742**

**04 noviembre 2017**

# Reporte del problema del agente viajero.

**En esta etapa se llevó a cabo la implementación de un algoritmo de aproximación usando el algoritmo de kruskal para obtener un árbol de expansión mínima y de él crear una solución aproximada al problema de agente viajero.**

## Definición del problema del agente viajero (PAV)

**El problema del agente viajero tiene como objetivo encontrar un recorrido que conecte a todos los nodos de un grafo, en este caso utilizamos como aplicación el hecho de conectar a diez ciudades visitando cada una de ellas y después volver a nuestro punto de partida, buscando minimizar la distancia de la ruta.**

## ¿Qué es lo difícil del PAV?

**Quizás lo difícil es la elección de la ruta más optima de entre todas las que el programa arroja.**

## ¿Qué hace un algoritmo de aproximación?

**Se vio que un algoritmo de aproximación minimiza la posibilidad de obtener un buen resultado en un periodo de tiempo reduciendo su complejidad.**

## ¿Qué hace el algoritmo de kruskal?

**El algoritmo de kruskal se encarga de encontrar el valor de la suma de todas las aristas o distancias entre los nodos de un grafo.**

## ¿Qué es un árbol de expansión mínima?

**Lo que yo entendí que era un árbol de expansión mínima es aquel compuesto por nuestros vértices y aristas, buscando aquella suma mínima de las aristas que conectan los vértices o nodos.**

## Descripción de ejemplo:

**Se realizó el problema del agente viajero con las siguientes ciudades y las distancias que existen entre ellas:**

## Código:

```
>>> from heapq import heappop,heappush
>>> from copy import deepcopy
>>> import random
>>> import time
>>> def permutation(lst):
        if len(lst)==0:
                return[]
        if len(lst)==1:
                return[lst]
        l=[]
        for i in range(len(lst)):
                m=lst[i]
                remlst=lst[:i]+lst[i+1:]
                for p in permutation(remlst):
                        l.append([m]+p)
                        return l


>>> class Fila:
        def __init__(self):
                self.fila = []
        def obtener(self):
                return self.fila.pop()
        def meter(self,e):
                self.fila.insert(0,e)
                return len(self.fila)
        @property
```

```
        def longitud(self):
                return len(self.fila)


>>> class Pila:
        def __init__(self):
                self.pila = []
        def obtener(self):
                return self.pila.pop()
        def meter(self,e):
                self.pila.insert(0,e)
                return len(self.pila)
        @property
        def longitud(self):
                return len(self.pila)


>>> 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 BFS(self,ni):
        visitados=[]
        f=Fila()
        f.meter(ni)
        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 DFS(self,ni):
        visitados=[]
        f=Pila()
        f.meter(ni)
        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 shortests(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
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 vecinoMasCercano(self):
```

```
        ni = random.choice(list(self.V))
        result=[ni]
        while len(result) < len(self.V):
            ln = set(self.vecinos[ni])
            le = dict()
            res =(ln-set(result))
            for nv in res:
                le[nv]=self.E[(ni,nv)]
            menor = min(le, key=le.get)
            result.append(menor)
            ni=menor
        return result


>>> g=Grafo()
>>> g.conecta('a','b',284)
>>> g.conecta('a','c',1286)
>>> g.conecta('d','c',1103)
>>> g.conecta('c','e',1733)
>>> g.conecta('f','e',1170)
>>> g.conecta('f','g',122)
>>> g.conecta('f','h',219)
>>> g.conecta('a','i',1483)
>>> g.conecta('d','j',316)
>>> g.conecta('k','e',831)
>>> g.conecta('a','d',913)
>>> print(g.kruskal())
```

```
>>> print([print(x,k.E[x]) for x in k.E])
>>>for r in range(10):
        ni=random.choice(list(k.V))
        dfs=k.DFS(ni)
        c=0
        print(dfs)
        print(len(dfs))
        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)
>>> data = list('abcdfghij')
>>> tim = time.clock()
>>> per = permutation(data)
>>> vm,rm=100000000000,[]
>>> for e in per:
        print(e)
        c=0
        for f in range(len(e)-1):
                c+=g.E[(e[f],e[f+1])]
        c+=g.E[(e[-1],e[0])]
        if c < vm:
                vm,rm = c,e
print(time.clock()-tim)
>>> print('minimo exacto',vm,rm)
```

**Monterrey a Cd. Victoria: 284 km**

**(‘a’) Monterrey a (‘b’) Acapulco: 1286 km**

**Monterrey a (‘c’) Tuxtla Gutiérrez: 1733 km**

**(‘d’) Cd. Victoria a Acapulco: 1103 km**

**Tuxtla Gutiérrez a Acapulco: 1170 km**

**Monterrey a (‘e’) China: 122 km**

**China a (‘f’) Saltillo: 219 km**

**Monterrey a (‘g’) La Paz: 1483 km**

**(‘h’) Ozuluama a Cd. Victoria: 316 km**

**(‘i’) Ciudad de México a Tuxtla Gutiérrez: 831 km**

**Monterrey a (‘j’) Cd. De México: 913 km**

## Soluciones:

**{('g', 'f'): 122, ('f', 'g'): 122, ('h', 'f'): 219, ('f', 'h'): 219, ('b', 'a'): 284, ('a', 'b'): 284, ('j', 'd'): 316, ('d', 'j'): 316, ('e', 'k'): 831, ('k', 'e'): 831, ('d', 'a'): 913, ('a', 'd'): 913, ('c', 'd'): 1103, ('d', 'c'): 1103, ('e', 'f'): 1170, ('f', 'e'): 1170, ('i', 'a'): 1483, ('a', 'i'): 1483, ('e', 'c'): 1733, ('c', 'e'): 1733}**

## Mejor solución:

**8174 : {'c', 'a', 'd', 'e', 'g', 'h', 'j', 'f', 'k', 'i', 'b'}**

## Heurística del vecino más cercano (¿Qué hace?)

**Consiste en:**

**1. Seleccionar al azar una ciudad como origen.**

**2. Visitar la ciudad más cercana aún no visitada.**

**3. Repetir hasta que todas las ciudades hayan sido visitadas.**