Dijkstra y Problema del viajero

Facultad de ciencias Físico - Matemáticas
Universidad Autonoma de Nuevo León
Sarai Elisabet Gómez Ibarra
1748263
Matemáticas computacionales

Mayo 2018

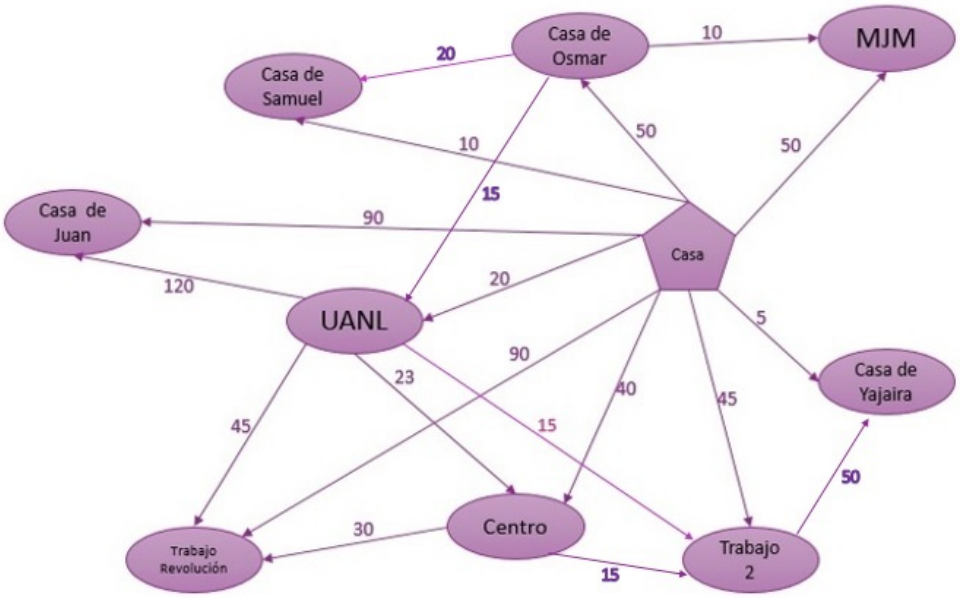
1 Grafo

Un grafo es una representación gráfica de diversos puntos que se conocen como nodos o vértices, los cuales se encuentran unidos a través de líneas que reciben el nombre de aristas; estos vértices tienes un valor y están unidos con los demás vértices por las aristas, que de igual manera tienen un valor numérico.

En Python el grafo está programado como:

```
class Grafo:
    def __init__(self):
        self.V = set()  # un conjunto
        self.E = dict()  # un mapeo de pesos de aristas
        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  # en ambos sentidos
        self.vecinos[v].add(u)
        self.vecinos[u].add(v)
```

El grafo que vamos a usar es el siguiente:



Donde el grafo en Phyton esta dado por:

```
    #Mi grafo
g= Grafo()
g.conecta('Casa','UANL',20)
g.conecta('Casa','CasaSamuel',15)
g.conecta('Casa','CasaOsmar',50)
g.conecta('Casa','MJM',50)
g.conecta('Casa','TrabajoR',90)
g.conecta('Casa','Centro',40)
g.conecta('Casa','Trabajo2',45)
g.conecta('Casa','CasaYajira',5)
g.conecta('Casa','CasaJuan',90)
g.conecta('CasaOsmar','MJM',10)
g.conecta('UANL','CasaJuan',120)
g.conecta('UANL','TrabajoR',45)
g.conecta('UANL','Centro',23)
g.conecta('UANL','Trabajo2',15)
g.conecta('Centro','Trabajo2',15)
g.conecta('Centro','TrabajoR',30)
g.conecta('Trabajo2','CasaYajaira',50)
```

```
g.conecta('CasaOsmar','UANL',15)
g.conecta('CasaSamuel','CasaOsmar',20)
```

Donde los valores que tienen representan el tiempo que se tarda en recorrer de ese vértice al otro.

2 Dijkstra

El método Dijkstra es un algoritmo que busca la forma más rápida de llegar de un nodo inicial a un nodo final, donde se tomando en cuenta el peso.

El algoritmo de Dijkstra en Python esta dado por:

```
from heapq import heappop, heappush
from copy import deepcopy

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

class Grafo:

    def __init__(self):
        self.V = set()  # un conjunto
        self.E = dict()  # un mapeo de pesos de aristas
        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  # en ambos sentidos
        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
    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):
            arista = t.pop()
            w = e[arista]
            del e[arista]
            (u, v) = arista
            c = comp.get(v, {v})
            if u not in 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
```

Donde tomaremos a mi Casa como vertice inicial nos da los siguientes resultados:

De la Casa a la Casa: (Casa, Casa; 0)
De la Casa a la CasaSamuel: (Casa, CasaSamuel; 10)
De la Casa a la CasaOsmar: (Casa, CasaOsmar; 50)
De la Casa a la CasaYajaira: (Casa, CasaYajaira; 5)
De la Casa a la CasaJuan: (Casa, CasaJuan; 90)

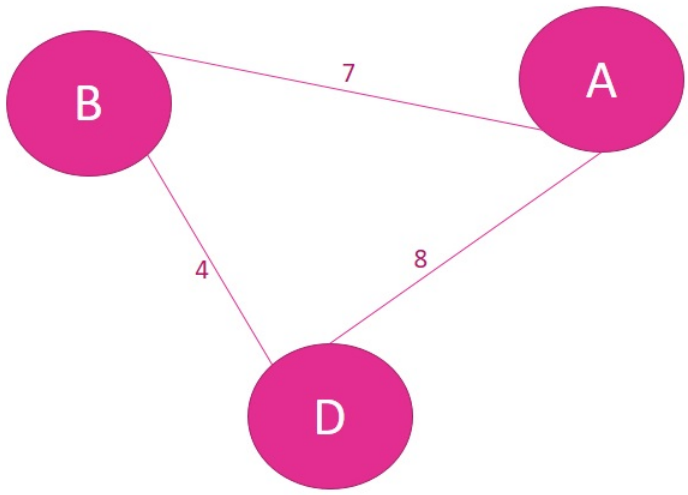
De la Casa a la Centro: (Casa, Centro; 40)
De la Casa a la TrabajoR: (Casa, UANL, TrabajoR; 65)
De la Casa a la MJM: (Casa, MJM; 50)
De la Casa a la Trabajo2: (Casa, UANL, Trabajo2; 35)
De la Casa a la UANL: (Casa, UANL; 20)

3 Problema del viajero

El problema del viajero o también conocido como TSP (Travelling Salesman Problem) es un problema matemático; el problema consiste en tener 'n' cantidad de destinos o nodos, en el cual una persona o 'el viajero' tiene que visitar todos los destinos sin repetir algún nodo, pero este recorrido que realice tiene que ser el más corto, barato o rápido dependiendo de lo que estén calculando.

A simple vista se ve como un problema muy sencillo, en cambio tiene muchos posibles, donde cuando el grafo tiene 'n' nodos, ¡entonces tiene n! posibles rutas, entonces es muy complicado encontrar la mejor ruta manualmente.

Un pequeño ejemplo seria, con el siguiente grafo:



Donde los posibles caminos están dados por n!

n=3 entonces son 6 los posibles caminos, que son:
A - B - C
A - C - B
B - C - A
B - A - C
C - B - A
C - A - B

Pero como es un grafo muy sencillo cualquiera puede ser un optimo camino, ya que cada uno de los caminos tiene la longiud de 19 unidades.

En Python el TSP está programado como:

```
def algorithm(cities):
    order = range(cities.shape[0])
    shuffle(order)
    length = calc_length(cities, order)
    start = time()

    changed = True
    while changed:

        changed = False

        for a in range(cities.shape[0]):
            for b in range(a + 1, cities.shape[0] - 1):
                c = choice(list(range(b + 1, cities.shape[0])))
                a, b, c = choice(list(permutations((a, b, c))))
                new_order = order[:a] + order[a:b][::-1] + order[b:c][::-1] + order[c:]
                new_length = calc_length(cities, new_order)
                if new_length < length:
                    length = new_length
                    order = new_order
                    changed = True
    return order, length


@jit
def calc_length(cities, path):
    length = 0
    for i in range(len(path)):
        length += dist_squared(cities[path[i - 1]], cities[path[i]])
```

```
return length
```

Donde nuestra mejor ruta calculada seria:
(CasaYajaira, Trabajo2, Centro, TrabajoR, UANL, CasaJuan, Casa, CasaSamuel, CasaOsmar, MJM)
Donde el grafo se veria de la siguiente manera: