Problema del agente viajero

Nombre: Alexandra Valeria Hernández Quintero **Matricula:** 1663507

Unidad de aprendizaje: Matemáticas Computacionales **Fecha:** 20/10/2017

Definición del problema del agente viajero:

Este problema consiste en encontrar la ruta más corta, pero no solo de un lugar a otro, sino más bien al tener varios lugares por visitar encontrar la forma de visitar a todos los lugares en el menor tiempo posible, los lugares pueden ser ciudades, estados, países, etc. Claro está que en la vida real no solo se toma en cuenta las distancias que hay de un lugar a otro ya que hay muchas variables, como el tráfico, el clima, el tipo de vehículo para viajar, cuanto tiempo durara en cada lugar o si surge algún contratiempo. Sin embargo por esta ocasión se despreciara ese tipo de variables y solo se centrara en las distancias de que hay de un lugar a otro.

¿Qué es lo difícil del problema del agente viajero (PAV)?

Aunque solo se centrara en la distancia, lo difícil de este problema es encontrar todas las rutas posibles que existan, por más que se parezca una ruta de otra, cualquier cambio mínimo en las distancias, ya es una ruta diferente y hay que tomarla en cuenta por esa podría ser la ruta más corta y/o eficaz. Si este problema se hiciera a mano se tardaría demasiado llegar a una conclusión, aun tomado en cuenta solo las distancias de un lugar a otro.

¿Qué es un algoritmo de aproximación?

Es un tipo de algoritmo generalmente usado para encontrar soluciones aproximadas es decir, lo más exacto posible o con errores mínimos, a problemas de optimización (por ejemplo el PAV), cuando se habla de optimización, comúnmente se refiere a buscar la mejor manera de realizar una actividad. Generalmente las soluciones que muestran este tipo de algoritmos son de calidad y cuyos tiempos de ejecución están acotados por cotas conocidas (están dentro de un rango establecido). Idealmente, la aproximación mejora su calidad para factores constantes pequeños (por ejemplo, dentro del 5% de la solución óptima).

¿Qué hace el algoritmo de kruskal?

Lo primero que hace kruskal es ordenar las aristas de grafo por su peso de menor a mayor, después el algoritmo kruskal intentara unir cada arista siempre y cuando no se forme un ciclo, es decir que no “pase” por la misma arista más de una vez. Como se ha ordenado las aristas por su peso entonces al principio se tomara la arista con el menor peso si los nodos que contienen a dicha arista no están en la misma componente conexa, entonces se unirán para formar una sola componente mediante la unión (ya antes mencionada), también se revisa si están o no en la misma componente conexa, ya que al hacer esto estamos evitando que se creen ciclos y que la arista que une dos vértices siempre sea la mínima posible.

Implementación de kruskal

```
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
```

¿Qué es un árbol de expansión mínima (MST)?

Dado un grafo conexo (todos los nodos están conectados entre sí), no dirigido G. Un árbol de expansión es un árbol compuesto por todos los vértices y algunas (posiblemente todas) de las aristas de G. Al ser creado un árbol no existirán ciclos, además debe existir una ruta entre cada par de vértices.

Un grafo puede tener muchos árboles de expansión, sin embargo cuando se habla de un árbol de expansión mínima se refiere a un árbol compuesto por todos los vértices y cuya suma de sus aristas es la de menor peso.

Descripción del ejemplo:

Un promotor quiere saber cuál es la ruta más corta, para ir a 10 lugares diferentes a entregar mercancía, los 10 lugares son:

1) COCA-COLA: Av. Alfonso Reyes 3001, Regina, 64290 Monterrey, N.L.

2) Cintermex: Av. Fundidora 501, Obrera, 64010 Monterrey, N.L.

3) UANL Campus Mederos: Avenida Lázaro Cárdenas S/n, Mederos, 64930 Monterrey, N.L.

4) FCFM - Facultad de Ciencias Físico Matemáticas: Pedro de Alba, Ciudad Universitaria, 66451 San Nicolás de los Garza, N.L.

5) Bosque Mágico Coca Cola: Eloy Cavazos s/n, La pastora, Sin Nombre de Col 33, 67140 Guadalupe, N.L.

6) Escuela industrial y preparatoria técnica Pablo Livas unidad poniente: Ocaso 100, Barrio Acero, 66050 Monterrey, N.L.

7) Escuela industrial y preparatoria técnica Pablo Livas unidad centro: José Mariano Jiménez 321 Sur, Centro, 64000 Monterrey, N.L.

8) Preparatoria No. 2: Mariano Matamoros 328, Obispado, Centro, 64060 Monterrey, N.L.

9) Preparatoria No. 9: Cd. del Maíz, Mitras Nte., 64300 Monterrey, N.L

10) Preparatoria 1 UANL (Unidad Apodaca): Elías Flores SN, Cabecera Municipal (Apodaca), Centro, 66600 Cd Apodaca, N.L.

Donde las distancias son las siguientes:

De 1 a 2	6.6
De 1 a 3	17.1
De 1 a 4	2.6
De 1 a 5	12.7
De 1 a 6	13.9
De 1 a 7	3.9
De 1 a 8	7.4
De 1 a 9	6.6
De 1 a 10	19

De 2 a 3	10.5
De 2 a 4	7-4
De 2 a 5	6.7
De 2 a 6	22.2
De 2 a 7	3.7
De 2 a 8	7.4
De 2 a 9	9.8
De 2 a 10	19.4

De 3 a 4	17
De 3 a 5	12.4
De 3 a 6	28
De 3 a 7	11.6
De 3 a 8	12.1
De 3 a 9	16.7
De 3 a 10	27.4

De 4 a 5	14
De 4 a 6	17.7
De 4 a 7	6.3
De 4 a 8	8.8
De 4 a 9	7.2
De 4 a 10	29.4

De 5 a 6	27.3
De 5 a 7	7.6
De 5 a 8	11
De 5 a 9	16.2
De 5 a 10	18.8

De 6 a 7	20
De 6 a 8	18
De 6 a 9	11.8
De 6 a 10	32.8
De 7 a 8	3.1
De 7 a 9	10.5
De 7 a 10	24.8
De 8 a 9	8.5
De 8 a 10	26.4
De 9 a 10	33.2

Implementación del algoritmo de aproximación usando kruskal

```
k=g.kruskal()
print(k)#Imprime el arbol de expansion minima
tim=time.clock()
mejor=-1
camino=[]
for r in range(10):
    ni=random.choice(list(k.V))
    dfs=k.DFS(ni)
    peso=0
    for i in range(len(dfs)-1):
        peso+=g.E[(dfs[i],dfs[i+1])]
    peso+=g.E[(dfs[-1],dfs[0])]

    print("Nodo inicial ",ni)
    for i in range(len(dfs)-1):
        print("De ",dfs[i],"\ta",dfs[i+1],"\tla distancia es ",g.E[(dfs[i],dfs[i+1])])
    print("De ",dfs[-1],"\ta ",dfs[0],"\tla distancia es ",g.E[(dfs[-1],dfs[0])])

    print("Costo total: ",peso,"\n")
    if mejor==-1 or mejor>peso:
        mejor=peso
        camino=dfs
print("La mejor ruta es la siguiente: ")
for k in camino:
    print(k,'->')
print(camino[0])
print("\nCon un costo de ",mejor)
print("Tiempo de ejecucion: ",time.clock()-tim)
print("\n-------------------------------------------------\n\n")
```

Resultados:

Después de hacer el respectivo grafo para el ejemplo e implementar el algoritmo de aproximación algunos de los resultados fueron los siguientes, donde cada número representa un lugar de los ya antes mencionados en el ejemplo y las distancias de un lugar a otro.

Solución 1		**Solución 2**		**Solución 3**	
Aristas	Distancias	Aristas	Distancias	Aristas	Distancias
9-6	11.8	9-1	6.6	8-7	3.1
6-1	13.9	1-4	2.6	7-2	3.7
1-4	2.6	4-7	6.3	2-3	10.5
4-7	6.3	7-8	3.1	3-5	12.4
7-2	3.7	8-2	7.4	5-10	18.8
2-3	10.5	2-3	10.5	10-1	19
3-5	12.4	3-5	12.4	1-9	6.6
5-10	18.8	5-10	18.8	9-6	11.8
10-8	26.4	10-6	32.8	6-4	17.7
8-9	8.5	6-9	11.8	4-8	8.8
Costo total:	**114.9**	**Costo total:**	**112.3**	**Costo total:**	**112.399**
Tiempo:	**0.0519s**	**Tiempo:**	**0.0601s**	**Tiempo:**	**0.0715s**

Solución 4		**Solución 5**	
Aristas	Distancias	Aristas	Distancias
5-2	6.7	1-9	6.6
2-7	3.7	9-6	11.8
7-8	3.1	6-4	17.7
8-1	7.4	4-7	6.3
1-4	2.6	7-8	3.1
4-9	7.2	8-2	7.4
9-6	11.8	2-3	10.5
6-3	28	3-5	12.4
3-10	27.4	5-10	18.8
10-5	18.8	10-1	19
Costo total:	**116.7**	**Costo total:**	**113.6**
Tiempo:	**0.03668s**	**Tiempo:**	**0.0679s**

Heurística del PAV (vecino más cercano):

¿Qué hace?

Pues bien de la forma más sencilla, el algoritmo al igual que nosotros en la vida real, al estar en un determinado lugar y tener una lista de lugares por visitar, buscara el lugar más cercano para visitarlo, claro está que el lugar más cercano dependerá mucho del lugar en el que estemos al inicio. En si lo que el algoritmo hace es tomar algún nodo (aleatorio), buscara el nodo más cercano al elegido inicialmente, ira hacia él, y se repetirá el mismo proceso hasta no tener más nodos no visitados.

Implementación:

```
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
```

Resultados:

Para estos resultados, al igual que en el algoritmo de aproximación, se hizo primero el grafo, pero a diferencia del algoritmo de aproximación, no usa kruskal como herramienta auxiliar, los resultados son los siguientes, donde cada número representa un lugar de los ya antes mencionados en el ejemplo y las distancias de un lugar a otro.

Solución 1		**Solución 2:**		**Solución 3:**	
Aristas	Pesos	Aristas	pesos	Aristas	Pesos
10-5	18.8	6-9	11.8	9-1	6.6
5-2	6.7	9-1	6.6	1-4	2.6
2-7	3.7	1-4	2.6	4-7	6.3
7-8	3.1	4-7	6.3	7-8	3.1
8-1	7.4	7-8	3.1	8-2	7.4
1-4	2.6	8-2	7.4	2-5	6.7
4-9	7.2	2-5	6.7	5-3	12.4
9-6	11.8	5-3	12.4	3-10	27.4
6-3	28	3-10	27.4	10-6	32.8
3-10	27.4	10-6	32.8	6-9	11.8
Costo total:	**116.699**	**Costo total:**	**117.100**	**Costo total:**	**117.1**
Tiempo:	**0.0305s**	**Tiempo:**	**0.0564s**	**Tiempo:**	**0.0568**

Solución 4:		**Solución 5:**	
Aristas	Pesos	Aristas	Pesos
5-2	6.7	2-7	3.7
2-7	3.7	7-8	3.1
7-8	3.1	8-1	7.4
8-1	7.4	1-4	2.6
1-4	2.6	4-9	7.2
4-9	7.2	9-6	11.8
9-6	11.8	6-5	27.3
6-3	28	5-3	12.4
3-10	27.4	3-10	27.4
10-5	18.8	10-2	19.4
Costo total:	**116.7**	**Costo total:**	**122.300**
tiempo	**0.0429s**	**Tiempo:**	**0.0356s**

Solución exacta

Para obtener la solución exacta es necesario calcular todas las formas posibles en las que se pueden permutar los distintos lugares (nodos), para esto haremos uso de la siguiente función:

```
def permutation(lst):
    if len(lst) == 0:
        return []
    if len(lst) == 1:
        return [lst]
    l = [] # empty list that will store current permutation
    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
```

Y después se utilizó el siguiente método de la clase Grafo:

```
def PAV(self):
    perm=permutation(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
```

Resultado:

Solución exacta:
1-10
10-5
5-3
3-2
2-7
7-8
8-6
6-9
9-4
4-1
Costo total: 107.09
Tiempo: 114.036s

Conclusiones:

Problema del agente viajero			
Algoritmo	Costo total:	Mejor ruta	Tiempo
Aproximación	112.3	9-1-4-7-8-2-3-5-10-6-9	0.0601s
Heurística	116.699	10-5-2-7-8-1-4- 9-6-3-10	0.0305s
Exacta	107.09	1-10-5-3-2-7-8-9-4-1	114.036s

Se puede observar que para tener la solución exacta tomo mucho más tiempo que tener una solución aproximada, la heurística fue la más alejada de la solución exacta pero le tomo mucho menos tiempo que al algoritmo de aproximación y el de la solución exacta para dar una posible solución, el algoritmo de aproximación sería muy útil cuando se quiere una solución aceptable y rápida, pero si se exige la exacta pues habrá que usar el algoritmo de la solución exacta.