Algoritmo de Kruskal

Luis Enrique López Nerio Universidad Autónoma de Nuevo León
luiselopeznerio@gmail.com

25 de Octubre del 2017

Abstract

Para este reporte se cubrirá otro tema del área de teoría de grafos, hasta el momento hemos visto diversos algoritmos que nos ayudan a encontrar información importante dentro de un grafo, nuestro siguiente objetivo será dado un grafo, encontrar todas las artistas del grafo que conecten a todos los vértices y que la suma de estos pesos sea mínima.
Para realizar esta tarea se introducirá el algoritmo de Kruskal, además se mencionaran algunas aplicaciones para este algoritmo.

I. Problema del agente viajero

Un grafo es un objeto matemático que contiene vértices y aristas, en si se puede definir matemáticamente como un dos conjuntos, el de vértices y el de aristas. El grafo puede ser ponderado o no ponderado, si es ponderado entonces a cada arista le corresponde un peso. Todas estas definiciones ya se introdujeron y se explicaron más a detalle en reportes anteriores.

Ahora, también habíamos dicho que un grafo es una abstracción de algo que observamos en la realidad, como una red de amigos, la red carretera de un país, etc. Por lo tanto plantearemos el siguiente problema, un agente vendedor quiere viajar por el país vendiendo sus productos, quiere visitar cada ciudad del país solamente una vez, pero también quiere minimizar la distancia recorrida y terminar en la ciudad donde empezó, la pregunta es, cual es el camino optimo a seguir.

Esto problema es llamado el problema del agente viajero y traducido a teoría de grafos se puede interpretar de la siguiente forma: dado un grafo ponderado $\mathcal{G} = (V, E)$ cual es el camino más corto que comienza en algún vértice, visita cada uno de los vértices al menos una vez y termina en el vértice inicial.

Este problema es un problema clásico dentro de las ciencias computacionales ya que se presenta en muchos procesos, además es considerado un problema "difícil", en las siguientes secciones veremos porque es un problema "dificil"

II. Porque el problema del agente viajero es "difícil'

El problema del agente viajero es considerado un problema difícil ya que computacionalmente no hay un algoritmo que puede resolverlo en un tiempo polinomio, esto quiere decir que de manera exacta no hay algoritmos que encuentren la solución en un tiempo razonable.

Imagine un grafo que representa a un país, cada vértice es una ciudad y digamos que hay 25 ciudades, cada una de ellas conectada con la otra, además cada arista con diferentes pesos,

para encontrar una solución exacta tendríamos que considerar todas la rutas posibles del grafo, esto quiere decir que hay $25! = 15,511,210,043,330,985,984,000,000$ rutas diferentes, además 25 son pocos nodos, aun y cuando nosotros optimizáramos el algoritmo un poco, este crece mucho más rápido comparado con la entrada.

Para tratar de evadir el problema se pueden encontrar algoritmos que nos den soluciones que no sean exactas pero que sean cercanas a la solución exacta, de esta manera sacrificamos exactitud contra tiempo computacional, a grandes rasgos existen dos grades tipos de algoritmos que nos ayudan a esto:

- Algoritmo de aproximación
- Heuristica

En las siguientes secciones se presentaran algunos algoritmos de aproximación y exactos para reslolver el problema del agente viajero.

III. Algoritmos de aproximación

Analizando el problema del agente viajero observamos que es un problema complicado de resolver, conforme el número de vertices crece, la complejidad computacional tambien lo hace, por lo tanto tal vez debemos encontrar otra forma de solucionar el problema, es aqui donde surgen los algoritmos de aproximación.

Un algoritmo de aproximación es aquel que nos da una solución cercana a la solución real con un cierto grafo de aproximación, estos algoritmos toman menos tiempo en la resolución del problema sacrificando exactitud contra tiempo. A continuación se presetnaran dos algoritmos de aproximación para el problema del agente viajero y uno de solución exacta.

IV. El arbol de expansión minima

Ahora se nos presentara un problema, que pasa si yo tengo un grafo, de este grafo que llamaremos original yo quiero un subgrafo, este subgrafo debe tener los mismo vértices que mi grafo original y debe tener aristas de tal manera que todos los vértices queden conectados pero sin que se presenten ciclos, esto es, del grafo original encuentra un grafo que contenga todas las aristas que conectan a todos los vértices y que la suma de los pesos sea mínima.

Este grafo que se genera del grafo original se llama árbol de expansión mínima, en la figura 1 se puede ver un ejemplo, las aristas resaltadas conectan todos los vértices del grafo original y la suma de las aristas es la mínima comparada con otros árboles de expansión para este mismo grafo. Ahora que definimos el problema, surge la pregunta, ¿Cómo encuentro el árbol de expansión minima de un grafo?, para responder esta pregunta introduciremos el algoritmo de Kruskal

V. Algoritmo de Kruskal

El algoritmo de Kruskal es un algoritmo que encuentra el arbol de expansión minima de un grafo, fue publicado por primera vez por Joseph Kruskal en 1956. La idea principal de este algoritmo es:

1 Se crea un grafo F, donde cada vertice del grafo es un árbol diferente
2 Se crea un conjunto S que contenga todos las aristas del grafo original

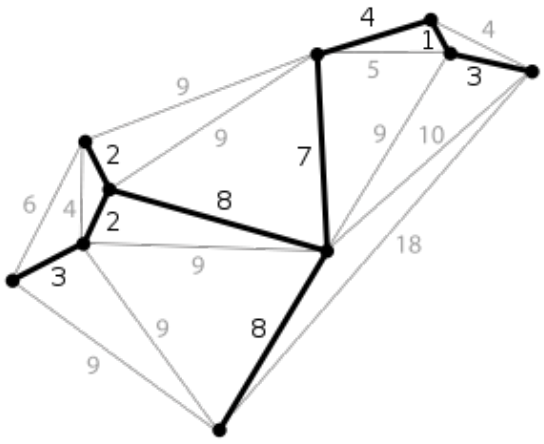

Figure 1: *ejemplo de un arbol de expansión minima*

3 Mientras S no este vacio y F no conecte a todos los vertices
3 Removemos la arista con peso minimo de S
4 Si la arista removida conecta dos difentes arboles de F encontes agregamos la arista a F, de esta manera dos arboles separados forman uno.

A continuación se presenta el codigo en python para el algoritmo 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):
        #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
```

Codigo 1. Algoritmo de Kruskal

Como podemos ver este algoritmo regresa como resultado el arbol de expansión minima.

Este algoritmo es importante ya que nos regresa un subgrafo del grafo que queriamos originalmente, que era un grafo que pasara por cada uno de los vertices al menos una vez y que la suma de los pesos fuera la minima posible. En las secciones siguinetes utlizaremos esta propiedad para encontrar una solución aproxiamda al problema del agente viajero.

VI. Vecino más cercano

El algoritmos del vecino mas cercano es un algoritmo de aproximación que nos ayuda a resolver el problema del agente viajero, al ser de aproximación no nos da una solución exacta sin embargo, por su facil interpretación y el tiempo en que nos da una solución, a veces es conveniente usarlo, el algortimos basicamente sigue los siguientes pasos:

1 Selecciona un vertice al azar y agregalo a una nueva lista
2 Del vertice seleccionado encuentra sus vecinos y encuentra el de menor peso que no este ya en la lista nueva, agregalo a la lista
3 Repite 2 hasta que la nueva lista contenga el mismo número de vertices que el grafo original
4 Agrega el vertice seleccionado al azar de nuevo pero al final de la lista

A continuación se presetna el codigo en python para realizar esta tarea.

```
1
2     def vecinoMasCercano(self):
3         lv = list(self.V)   ##lista de vertices
4         random.shuffle(lv)
5         ni = lv.pop()
6         inicial = ni
7         lv2=list()
8         lv2.append(ni)
9         peso=0
10        while len(lv2)<len(self.V):
11            le = list()
12            ln=list()
13            ln = self.vecinos[ni]
14            for nv in ln:
15                if nv not in lv2:
16                    le.append((nv,self.E[(ni,nv)]))
17            sorted(le, key = lambda le: le[1] )
18            t=le[0]
19            lv2.append(t[0])
20            peso=peso+t[1]
21            ni=t[0]
22        peso=peso+self.E[lv2[-1], inicial]
23        lv2.append(inicial)
24        return (lv2,peso)
```

Codigo 2. Algoritmo del vecino más cercano

VII. Solución exacta

Para encontrar una solución exacta para el problema del agente viajero se podria hacer los siguiente, teniendo n vertices calculamos todos los caminos posibles que se pueden formar, esto seria encontrar $n!$ caminos, entonces nuestra tarea es encontrar todas las permutaciones posibles, a continuación se presenta un algoritmo que puede hacer esto:

```
1
2 def permutation(lst):
3     if len(lst) == 0:
4         return []
5     if len(lst) == 1:
6         return [lst]
7     l = [] # empty list that will store current permutation
8     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
```

Codigo 3.- Algoritmo para encontrar permutaciones

VIII. Experimento

Para probar nuestros algoritmos se realizara un experimento, de la ciudad de Monterrey tomaremos 10 lugares turisticos, estos lugares turisticos seran considerados los vertices de un grafo, de cada lugar turistico calcularemos el tiempo que nos toma en promedio llegar a cada unos de los otros, el tiempo promedio entre dos lugares sera el peso de la arista que los une.

Los 10 lugares que eligi son los siguientes:

- Cola de Caballo
- Faro de Comercio
- Chipinque
- Grutas de Garcia
- Fundidora
- Cadereyta
- Basilica
- Cerro de la Silla
- Cerro de las Mitras
- Estadio BBVA

Ahora, la idea es encontrar un camino que comienze en alguno de estos vertices y que se vaya moviendo a cada uno de los otros sin pasar por uno que ya haya visitado, para finalmente llegar al vertice inicial, este camino también recibe el nombre de Ciclo Hamiltoniano.

Naturalmente se encontraran diferentes ciclos en el grafo, lo importante es tratar de encontrar el que nos haga pasar por aristas de tal manera que la suma de los pesos de las aristas visitadas sea la menor posible. Para realizar esta tarea utilizaremos nuestro algoritmos de Kruskal ademas del siguiente codigo:

```
 m = grafo()
m.conecta('Cola de Caballo', 'Faro de Comercio', 46)
m.conecta('Cola de Caballo', 'Chipinque', 69)
m.conecta('Cola de Caballo', 'Grutas de Garcia', 91)
m.conecta('Cola de Caballo', 'Fundidora', 47)
m.conecta('Cola de Caballo', 'Cadereyta', 63)
m.conecta('Cola de Caballo', 'Basilica', 48)
m.conecta('Cola de Caballo', 'Cerro de la Silla', 61)
m.conecta('Cola de Caballo', 'Cerro de las Mitras', 66)
m.conecta('Cola de Caballo', 'Estadio BBVA', 50)

m.conecta('Faro de Comercio','Chipinque', 30)
m.conecta('Faro de Comercio','Grutas de Garcia', 49)
m.conecta('Faro de Comercio','Fundidora', 7)
m.conecta('Faro de Comercio','Cadereyta', 38)
m.conecta('Faro de Comercio','Basilica', 5)
m.conecta('Faro de Comercio','Cerro de la Silla', 24)
m.conecta('Faro de Comercio','Cerro de las Mitras',23)
m.conecta('Faro de Comercio','Estadio BBVA', 13)
```

```
20
21 m.conecta('Chipinque','Grutas de Garcia',66 )
22 m.conecta('Chipinque','Fundidora', 30)
23 m.conecta('Chipinque','Cadereyta', 62)
24 m.conecta('Chipinque','Basilica', 28)
25 m.conecta('Chipinque','Cerro de la Silla', 48)
26 m.conecta('Chipinque','Cerro de las Mitras', 41)
27 m.conecta('Chipinque','Estadio BBVA', 37)
28
29 m.conecta('Grutas de Garcia','Fundidora', 51)
30 m.conecta('Grutas de Garcia','Cadereyta', 81)
31 m.conecta('Grutas de Garcia','Basilica',49 )
32 m.conecta('Grutas de Garcia','Cerro de la Silla', 68)
33 m.conecta('Grutas de Garcia','Cerro de las Mitras', 48)
34 m.conecta('Grutas de Garcia','Estadio BBVA', 56)
35
36 m.conecta('Fundidora','Cadereyta', 34)
37 m.conecta('Fundidora','Basilica',8 )
38 m.conecta('Fundidora','Cerro de la Silla', 22)
39 m.conecta('Fundidora','Cerro de las Mitras', 23)
40 m.conecta('Fundidora','Estadio BBVA', 8)
41
42 m.conecta('Cadereyta','Basilica', 38)
43 m.conecta('Cadereyta','Cerro de la Silla', 43)
44 m.conecta('Cadereyta','Cerro de las Mitras', 56)
45 m.conecta('Cadereyta','Estadio BBVA', 34)
46
47 m.conecta('Basilica','Cerro de la Silla', 26)
48 m.conecta('Basilica','Cerro de las Mitras', 25)
49 m.conecta('Basilica','Estadio BBVA', 15 )
50
51 m.conecta('Cerro de la Silla','Cerro de las Mitras', 45)
52 m.conecta('Cerro de la Silla','Estadio BBVA', 19)
53
54 m.conecta('Cerro de las Mitras','Estadio BBVA', 32)
55
56 k = m.kruskal()
57 for r in range(50):
58     ni = random.choice(list(k.V))
59     dfs =  k.DFS(ni)
60     c = 0
61     #print(dfs)
62     #print(len(dfs))
63     for f in range(len(dfs) -1):
64             c += m.E[(dfs[f],dfs[f+1])]
65             print(dfs[f], dfs[f+1], m.E[(dfs[f],dfs[f+1])] )
66
67     c += m.E[(dfs[-1],dfs[0])]
68     print(dfs[-1], dfs[0], m.E[(dfs[-1],dfs[0])])
69     print('costo',c,'\n')
```

Codigo 4. Codigo para encontrar el ciclo menor

Al correr el codigo nos encontramos que el camino con el costo menor es el siguiente: Chipinque Basilica 28 Basilica Faro de Comercio 5 Faro de Comercio Cola de Caballo 46 Cola de Caballo Fundidora 47 Fundidora Estadio BBVA 8 Estadio BBVA Cerro de la Silla 19 Cerro de la Silla Cadereyta 43 Cadereyta Cerro de las Mitras 56 Cerro de las Mitras Grutas de Garcia 48 Grutas de Garcia Chipinque 66 costo 366

Entonces el camino que recorre todos los vertices y que tiene un tiempo menor comienza en Chipinque y le toma 6 horas y 6 minutos visitar cada uno de los vertices.

Utilizando el algoritmo del vecino mas cercano se encontraron multiples soluciones, la mejor minimizaba el peso de las aristas es la siguiente:

(['Cadereyta', 'Grutas de Garcia', 'Chipinque', 'Basilica', 'Fundidora', 'Cerro de las Mitras', 'Estadio BBVA', 'Cola de Caballo', 'Faro de Comercio', 'Cerro de la Silla', 'Cadereyta'], 401)

Tamien se utlizo el algoritmo para encontar las permutaciones, tan solo para calcular las permutaciones el algoritmo se tardo 99 segundos, aun sin probar cada una de las rutas para encontrar el minimo.

IX. Conclusión

Mi conclusión es que el algoritmo de Kruskal nos da un abanico de posibilidades en las cuales podemos aplicar nuestra clase de Grafos. Ademas nos permite encontar una solución mucho mas rapido que si trataramos de encontrarla de forma exacta. Los algortimos de aproximación son el futuro hasta que no encontremos algoritmos exactos que de alguna manera encunetren la solución de manera mas rapida.