Reporte de Algoritmo de Kruskal

Haydee Judith Arriaga Ponce

Matricula: 1659539

l.judithmat@gmail.com

https://github.com/haydee10arriaga/1659539_Mat.Com

3 de Noviembre de 2017

1. PROBLEMA DEL AGENTE VIAJERO

De manera simple, se describe como un agente que debe visitar una vez a cada ciudad de entre un conjunto y regresar a su punto de partida formando un tour, de tal forma que el recorrido total sea mínimo. Esto es lo que de manera general conocemos como ciclo Hamiltoniano, a continuación, desarrolla de manera más formal el modelo del TSP.

- Formulación del TSP

Un agente viajero tiene que viajar a n ciudades (1, 2,…,n), el costo de viajar de la ciudad i a la ciudad j es cij, con i□ j para toda n; además debe iniciar en una ciudad, visitar cada una de las demás ciudades exactamente una vez en cierto orden y al final regresar a la ciudad inicial. El problema es determinar el orden en el que debe viajar a través de las distintas ciudades minimizando el costo total.

Suponga que inicia en la ciudad 1 si viaja a las ciudades en orden de i a i+1, desde i=1, hasta i=n-1, y después de la ciudad n a la ciudad 1, esta rutina se puede representar como “1,2,3,…,n+1” tal orden es conocido como tour. De modo que un tour es un circuito que sale una sola vez de cada ciudad. Por tanto la ciudad inicial es inmaterial y sin pérdida de generalidad podemos decir que se puede elegir cualquier ciudad para que sea la ciudad inicial y de ahí podemos seguir a cualesquiera otra ciudad n-1, así que hay n-1 formas diferentes de escoger la ciudad que sucede a la primera elección y de ésa ciudad se puede viajar a las n-2 ciudades restantes, etc. De modo que el número total de posibles tours en un problema de n ciudades es (n-1)(n-2)…1=(n-1)!.

Supongamos que t es un tour dado, se definen las variables enteras como:

$$x_{ij} = \begin{cases} 1 \text{ si el agente viaja de la ciudad } i \text{ a la ciudad } j \text{ en el tour } t \\ 0 \text{ de otra forma} \end{cases}$$

Entonces X=(xij) es obviamente una asignación y tal asignación corresponde al tour t, pero hay asignaciones que no son tours por ejemplo: podemos ver que la asignación {(1,2),(2,1),(3,4),(4,3)} no es un tour porque los trayectos en esta asignación no forman un circuito único de todas las ciudades, pero sí forman dos subtours (uno entre la ciudad 1 y 2 y otro entre la ciudad 3 y 4) lo cual no es posible para el problema que estudiamos.
Contemplando todas las consideraciones expuestas el planteamiento
matemático del problema del agente viajero es:

$$\begin{array}{lll} F.O. & \text{Minimizar } \sum_i \sum_j c_{ij} x_{ij} & \\ \text{Sujeto a:} & & \\ & \sum_i x_{ij} = 1 & \text{para toda } j \\ & \sum_j x_{ij} = 1 & \text{para toda } i \\ & u_i - u_j + (n+1) x_{ij} \leq n & \text{para } i=0,1,2,n;\ j=1,2,3,\ldots n+1;\ i \neq j \\ & x_{ij} = \{0,\ 1\} & \text{para toda } i,\ j \\ & X = (x_{ij}) & \text{es un tour asignado} \end{array}$$

La primera desigualdad fuerza a que el agente salga sólo una vez de cada ciudad, la segunda lo fuerza a que llegue sólo una vez a cada ciudad, la combinación de estas dos restricciones asegura que sólo se visite una vez cada ciudad.
Este modelo es general y existen muchas variantes, en particular para evitar la formación de subtours, lo cual genera una gran cantidad de formulaciones y métodos o estrategias de solución en función del tamaño o instancia del problema, es decir, del número de ciudades que debe visitar el agente, anotando que a mayor instancia se incrementa también la complejidad de solución.

2. PORQUE EL PROBLEMA DEL AGENTE VIAJERO ES “DIFICIL”

Es dificil ya que de manera exacta no hay algoritmos que encuentren la solucion en un tiempo razonable, ya que computacionalmente no hay un algoritmo porque para dicho problema tiene que ser estudiado el grado de complejidad y posible soluciones.

Es decir, es necesario abordar el tema de complejidad computacional, que estudia la “dificultad” de problemas de importancia teórica y práctica, como es el caso del TSP.

El objetivo fundamental de la Complejidad Computacional es clasificar los problemas de acuerdo a su manejabilidad, es decir, ¿es posible obtener una solución tomando él o los algoritmos más eficientes para resolverlos?, o bien, se quiere determinar las respuestas a las siguientes preguntas:

¿Qué tan manejable es el problema?

Si el problema es manejable, ¿es eficiente el algoritmo?

En general, los distintos grados de complejidad son subjetivos (varían considerablemente de acuerdo al modelo computacional, a los recursos disponibles, a las variantes de las estructuras de datos, etc.). Por lo tanto, un objetivo primario del estudio de la complejidad es definir cuáles problemas son tratables, y cuáles no.

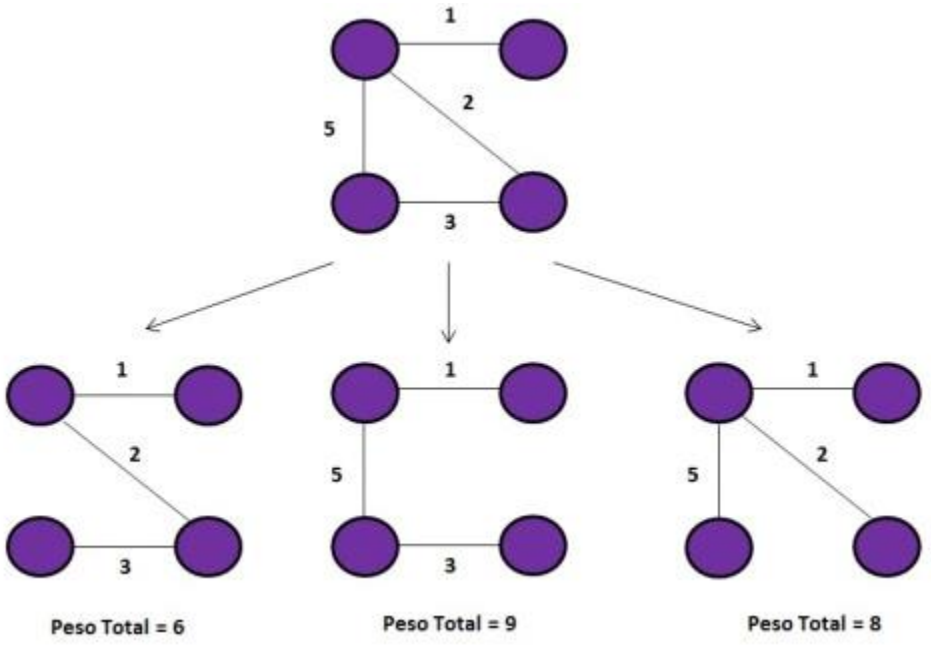
Por ejemplo, se puede afirmar que los problemas de programación lineal son tratables, es decir, tienen solución óptima aún para instancias grandes. En cambio, hay problemas que no son tratables, como el TSP, que en la práctica sólo se resuelve analíticamente para instancias pequeñas. Para tratar de evadir 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

3. EL ARBOL DE EXPANSIÓN MINIMA

Este problema surge cuando todos los nodos de una red deben conectarse entre ellos. El árbol de expansión mínima es apropiado para problemas en los cuales la redundancia es expansiva, el flujo a lo largo de los arcos se considera instántaneo. Este problema se refiere a utilizar las ramas o arcos de la red para llegar a todos los nodos de la red, de manera tal que se minimiza la longitud total.

Dado un grafo conexo, no dirigido y con pesos en las aristas, un árbol de expansión mínima es un árbol compuesto por todos los vértices y cuya suma de sus aristas es la de menor peso. Al ejemplo anterior le agregamos pesos a sus aristas y obtenemos los arboles de expansiones siguientes:



De la imagen anterior el árbol de expansión mínima seria el primer árbol de expansión cuyo peso total es 6.

El problema de hallar el Árbol de Expansión Mínima (MST) puede ser resuelto con varios algoritmos, los mas conocidos con Prim y Kruskal (el ponemos en practica en este reporte) ambos usan técnicas voraces (greedy).

4. ALGORITMO DE KRUSKAL

Este algoritmo fue escrito por Joseph Kruskal y publicado en 1956, es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor total de todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa). El algoritmo de Kruskal es un ejemplo de algoritmo voraz.

DESCRIPCION

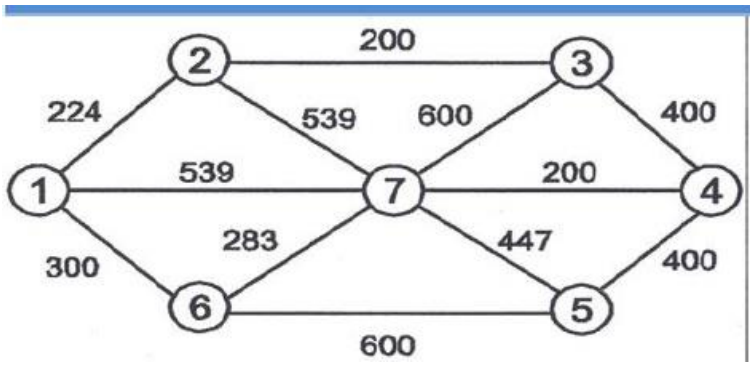
- Se crea un bosque *B* (un conjunto de árboles), donde cada vértice del grafo es un árbol separado

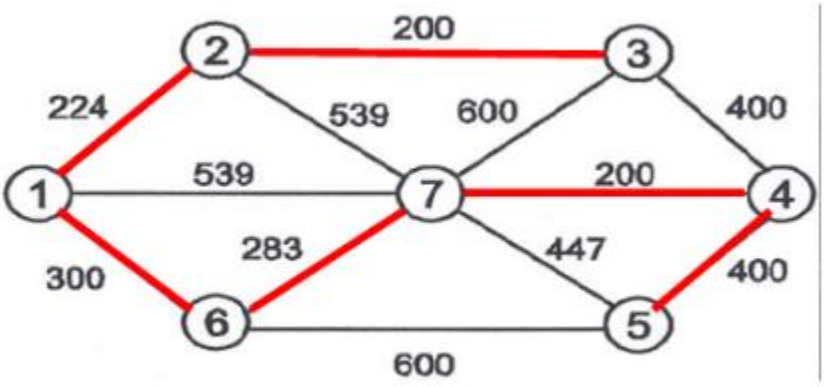
- Se crea un conjunto *C* que contenga a todas las aristas del grafo
- Mientras *C* es *no vacío*
  - eliminar una arista de peso mínimo de *C*
  - si esa arista conecta dos árboles diferentes se añade al bosque, combinando los dos árboles en un solo árbol
  - en caso contrario, se desecha la arista

Al acabar el algoritmo, el bosque tiene un solo componente, el cual forma un árbol de expansión mínimo del grafo.

Ejemplo 1:

Dada la siguiente red obtenga el árbol de expansión mínima.





Árbol de expansión mínima = 200+224+300+283+200+400 = 1607

La ruta marcada con rojo es la mejor la optima despues de haber checado las demas.

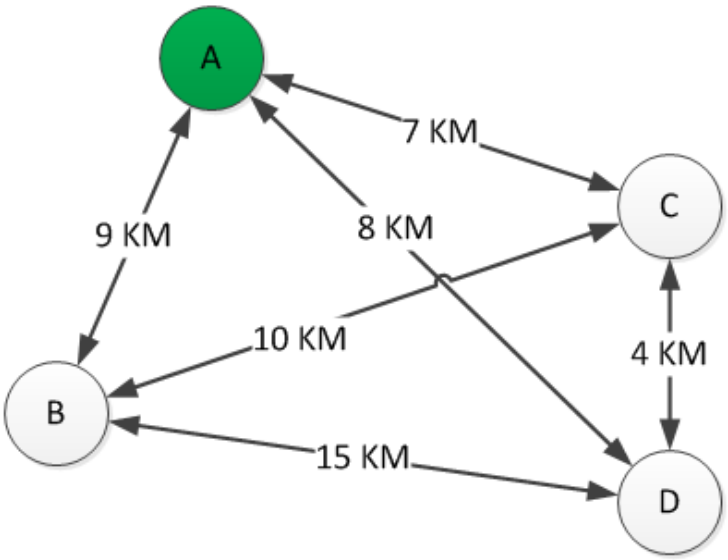
A continuacion se presenta el codigo en python para el algoritmo de Kruskal:

```
117     def kruskal(self):
118         e = deepcopy(self.E)
119         arbol = Grafo()
120         peso = 0
121         comp = dict()
122         t = sorted(e.keys(), key = lambda k: e[k], reverse=True)
123         nuevo = set()
124         while len(t) > 0 and len(nuevo) < len(self.V):
125             #print(len(t))
126             arista = t.pop()
127             w = e[arista]
128             del e[arista]
129             (u,v) = arista
130             c = comp.get(v, {v})
131             if u not in c:
132                 #print('u ',u, 'v ',v ,'c ', c)
133                 arbol.conecta(u,v,w)
134                 peso += w
135                 nuevo = c.union(comp.get(u,{u}))
136                 for i in nuevo:
137                     comp[i]= nuevo
138         print('MST con peso', peso, ':', nuevo, '\n', arbol.E)
139         return arbol
140
```

5. VECINO MÁS CERCANO

El método del vecino más cercano es un algoritmo heurístico diseñado para solucionar el problema del agente viajero, no asegura una solución óptima, sin embargo suele proporcionar buenas soluciones, y tiene un tiempo de cálculo muy eficiente. El método de desarrollo es muy similar al utilizado para resolver problemas de árbol de expansión mínima.

EJEMPLO:



El método consiste en una vez establecido el nodo de partida, evaluar y seleccionar su vecino más cercano. En este caso:

Vecinos de A	B	C	D
Distancia	9	7	8

En la siguiente iteración habrá que considerar los vecinos más cercanos al nodo C (se excluye A por ser el nodo de origen):

Vecinos de C	B	D
Distancia	10	4

En la siguiente iteración los vecinos más cercanos de D serán C, con quien ya tiene conexión, A quién es el nodo de origen y B, por esta razón B se debe seleccionar por descarte. Al estar en B todos los nodos se encuentran visitados, por lo que corresponde a cerrar la red uniendo el nodo B con el nodo A, así entonces la ruta solución por medio del vecino más próximo sería A, C, D, B, A = 7, 4, 15, 9 = 35 km.

Este es un caso en el que a pesar de tener una red compuesta por pocos nodos, el método del vecino más cercano no proporciona la solución óptima, la cual calculamos con el método de fuerza bruta como 31 km.

A continuacion se presenta el codigo en python para realizar esta tarea:

```
141     def vecinoMasCercano(self):
142         lv = list(self.V)
143         random.shuffle(lv)
144         ni = lv.pop()
145         le = dict()
146         while len(lv)>0:
147             ln = self.v[ni]
148             for nv in ln:
149                 le[nv]=self.E[(ni,nv)]
150             menor = min(le.values())
151             lv.append(menor)
152             del lv[menor]
153         return lv
```

6. SOLUCION EXACTA

La solución más directa puede ser, intentar todas las permutaciones (combinaciones ordenadas) y ver cuál de estas es la menor (usando una Búsqueda de fuerza bruta). El tiempo de ejecución es un factor polinómico de orden {O(n!)}, el Factorial del número de ciudades, esta solución es impracticable para dado solamente 20 ciudades.

Entonces teniendo n vertices calculamos todos los caminos posibles que se pueden formar, esto seria encontrar n! caminos, entonces la tarea es encontrar todas las permutaciones posibles, se presenta a continuacion el algoritmo que puede realizar lo anteriormente mencionado:

```
import time
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

```

7. EXPERIMENTO

Se realizara un experimento para probar nuestros algoritmos, del estado Nuevo León tomaremos el municipio de Guadalupe como el punto de partida de un turista que quiere partir hacia otros municipios de Nuevo León. Se diseño un código muy completo para optimizar las rutas de su traslado por lo cual buscaremos la ruta más corta para llegar a los municipios de modo que sean visitados todos una vez sin repetir y regresar a su punto de partida, de tal forma que el recorrido total sea minimo.

Los municipios seran considerados los vertices de un grafo que son

- ✓ Guadalupe
- ✓ Apodaca
- ✓ Juarez
- ✓ Monterrey
- ✓ García
- ✓ San Nicolas de los Garza
- ✓ Allende
- ✓ Escobedo
- ✓ Linares
- ✓ Pesqueria

Lo mas normal es encontrar diferentes ciclos en el grafo, pero lo importante es 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 practica utilizamos el algoritmo Kruskal ademas de este codigo con la informacion requerida:

```
m= Grafo()
m.conecta('Guadalupe', 'Apodaca',16)
m.conecta('Guadalupe', 'Juarez',24)
m.conecta('Guadalupe', 'Monterrey',6)
m.conecta('Guadalupe', 'Garcia',43)
m.conecta('Guadalupe', 'San Nicolas',11)
m.conecta('Guadalupe', 'Allende',60)
m.conecta('Guadalupe', 'Escobedo',19)
m.conecta('Guadalupe', 'Linares',131)
m.conecta('Guadalupe', 'Pesqueria',29)

m.conecta('Apodaca','Juarez',26)
m.conecta('Apodaca','Monterrey',20)
m.conecta('Apodaca','Garcia',46)
m.conecta('Apodaca','San Nicolas',14)
m.conecta('Apodaca','Allende',75)
m.conecta('Apodaca','Escobedo',23)
m.conecta('Apodaca','Linares',147)
m.conecta('Apodaca','Pesqueria',16)

m.conecta('Juarez','Monterrey',31)
m.conecta('Juarez','Garcia',74)
m.conecta('Juarez','San Nicolas',36)
m.conecta('Juarez','Allende',50)
m.conecta('Juarez','Escobedo',44)
m.conecta('Juarez','Linares',90)
m.conecta('Juarez','Pesqueria',32)

m.conecta('Monterrey','Garcia',37)
m.conecta('Monterrey','San Nicolas',9)
m.conecta('Monterrey','Allende',59)
m.conecta('Monterrey','Escobedo',15)
m.conecta('Monterrey','Linares',130)
m.conecta('Monterrey','Pesqueria',36)
m.conecta('Garcia','San Nicolas',51)
m.conecta('Garcia','Allende',93)
m.conecta('Garcia','Escobedo',34)
m.conecta('Garcia','Linares',165)
m.conecta('Garcia','Pesqueria',65)

m.conecta('San Nicolas','Allende',66)
m.conecta('San Nicolas','Escobedo',8)
m.conecta('San Nicolas','Linares',136)
m.conecta('San Nicolas','Pesqueria',38)

m.conecta('Allende','Escobedo',76)
m.conecta('Allende','Linares',72)
m.conecta('Allende','Pesqueria',76)

m.conecta('Escobedo','Linares',148)
m.conecta('Escobedo','Pesqueria',36)

m.conecta('Linares','Pesqueria',148)




k = m.kruskal()
for r in range(50):
    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 += m.E[(dfs[f],dfs[f+1])]
            print(dfs[f], dfs[f+1], m.E[(dfs[f],dfs[f+1])] )

    c += m.E[(dfs[-1],dfs[0])]
    print(dfs[-1], dfs[0], m.E[(dfs[-1],dfs[0])])
    print('costo',c,'\n')


```

Al ejecutar el código encontramos el camino con menor costo, el menor peso.

La ruta más óptima para visitar los municipios es:

MST con peso 233: {'Guadalupe', 'Apodaca', 'Monterrey', 'San Nicolás', 'Linares', 'Escobedo', 'Juárez', 'Allende', 'García', 'Pesquería'}

{('Guadalupe', 'Monterrey'): 6, ('Escobedo', 'San Nicolás'): 8, ('Monterrey', 'San Nicolas'): 9, ('San Nicolás', 'Apodaca'): 14, ('Apodaca', 'Pesquería'): 16, ('Juárez', 'Guadalupe'): 24, ('Escobedo', 'García'): 34,('Allende', 'Juárez'): 50,('Linares', 'Allende'): 72, }

Municipio	Municipio	Peso
Guadalupe	Monterrey	6
Escobedo	San Nicolás	8
Monterrey	San Nicolás	9
San Nicolás	Apodaca	14
Apodaca	Pesquería	16
Juárez	Guadalupe	24
Escobedo	García	34
Allende	Juárez	50
Linares	Allende	72

Después la implementación del algoritmo del vecino más cercano se encontraron varias soluciones, las mejores 3 son:

Municipio I.	Municipio F.	Peso
Juárez	Guadalupe	24
Guadalupe	Monterrey	6
Monterrey	San Nicolás	9
San Nicolás	Escobedo	8
Escobedo	García	34
García	Apodaca	46
Apodaca	Pesquería	16
Pesquería	Allende	76
Allende	Linares	72
Linares	Juárez	90

Con un costo de: 381

Municipio I.	Municipio F.	Peso
Linares	Allende	72
Allende	Juárez	50
Juárez	Guadalupe	24
Guadalupe	Monterrey	6
Monterrey	San Nicolás	9
San Nicolás	Escobedo	8
Escobedo	García	34
García	Apodaca	46
Apodaca	Pesquería	16
Pesquería	Linares	148

Con un costo de: 413

Municipio I.	Municipio F.	Peso
Apodaca	Pesquería	16
Pesquería	San Nicolás	38
San Nicolás	Monterrey	9
Monterrey	Guadalupe	6
Guadalupe	Juárez	24
Juárez	Allende	50
Allende	Linares	72
Linares	Escobedo	148
Escobedo	García	34
García	Apodaca	46

Con un costo de: 443

El tiempo de ejecución de todo el código en realizar las permutaciones es de 56.68199582367478.

Observaciones:

La inserción aleatorizada ofrece soluciones de buena calidad aun en instancias grandes, pero es más tardada que el vecino más cercano.

La búsqueda local unida al vecino más cercano genera una mejora significativa sin comprometer el tiempo. Unida a la inserción aleatorizada, sigue obteniendo muy buenas soluciones, sin embargo, el tiempo aumenta considerablemente.

CONCLUSION

Al llevar a cabo esta práctica y con la información recabada me di cuenta de que el problema del agente viajero a pesar de todo el tiempo que ha sido analizado para poder optimizarlo hasta llegar a obtener la mejor solución posible no se ha logrado la exactitud en tiempo ya que como se muestra en esta práctica al correr el código encontramos las posibles mejores respuestas o rutas al problema planteado mediante los métodos y algoritmos implementados nos arrojó las que generan menor peso y costo. Considero que para poder decir que algoritmo utilizar realmente se debe ponderar tanto el tiempo que se tiene para dar una solución, así como qué tan estricta es la exigencia de la calidad de la solución.

Realmente considero que ha como va avanzando la tecnología a pasos agigantados se lograra llegar realmente a la exactitud a solución de este problema ya que cada vez hay más contribución en las soluciones del TSP.

Me parece que a pesar de los pequeños detalles es de gran ayuda para resolver ciertos problemas que se nos presenten en lo particular me ayudo a comprender mejor este problema del agente viajero que lo acababa de ver en la clase de modelado matemático me pareció excelente que el maestro nos encargara llevar a cabo esta práctica ya que lo puedo implementar ejercicios que tenga en modelado matemático sobre este caso en particular del agente viajero o simplemente es interesante indagar más por mi cuenta en cuestión general de la programación ya que como matemática me cierro mucho en cuestión de aprender o ampliar mi conocimiento sobre programación.