Algoritmo Kruskal

Reporte para la clase de Matemáticas Computacionales

Paulina Aguirre García | 1837503

4 de noviembre de 2017

Introducción. –

A lo largo del curso de Matemáticas Computacionales fuimos instruidos en el área de Teoría de Grafos, una rama muy interesante de las Matemáticas, la cual puede ser vista desde una perspectiva más computacional. Para esto, hemos tratado con diversos algoritmos con características particulares que nos ayudan a recorrer un grafo. En este reporte trataremos uno de los problemas de optimización más estudiados, haciendo uso de un nuevo algoritmo.

Además de incluir un ejercicio de este problema.

A. Problema del agente viajero.

El problema del agente viajero (**TSP** *por sus siglas en inglés*), responde a la siguiente pregunta: dada una lista de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita a cada cuidad exactamente una vez y al final regresa a la cuidad origen? El concepto de “ciudad” representa, por ejemplo: clientes o algún punto de reunión y el concepto de “distancia” representa el tiempo de viaje o costo. Es decir, tenemos un grafo ponderado, puesto que hay distancias entre las ciudades (peso) y queremos recorrer dicho grafo, donde las ciudades o clientes serán los llamados nodos.

Este es un problema **NP-hard** dentro de la optimización combinatoria, muy importante en la investigación de operaciones y la ciencia de la computación.

El problema del agente viajero fue definido en los años 1800s por el matemático irlandés W. R. Hamilton y por el matemático británico Thomas Kirkman.

B. Dificultad.

El problema del agente viajero es considerado difícil ya que la versión de decisión del TSP pertenece a la clase de los problemas **NP-completos**, por tanto, es probable que en el caso peor el tiempo de ejecución para cualquier algoritmo que resuelva el TSP aumente de forma exponencial con respecto al número de ciudades.

Sea t un estado que cuente con 20 ciudades, tal que la distancia entre cada una de ellas varia respecto a las otras. Si quisiéramos recorrer este supuesto t estado en el menor tiempo posible, tendríamos que considerar todas las rutas posibles para realmente obtener la ruta más corta; es decir, $20! = 2\,432\,902\,008\,176\,640\,000$ rutas. Y esto es verdaderamente absurdo. Para tratar de evadir este problema existen algoritmos que brindan una solución no exacta pero muy cercana a la exactitud, estos son:

- Algoritmos de aproximación
- Heurística

*En el problema se presentan **N!** rutas posibles, es decir el número de permutaciones de las ciudades que tengamos.*

C. Algoritmos de aproximación.

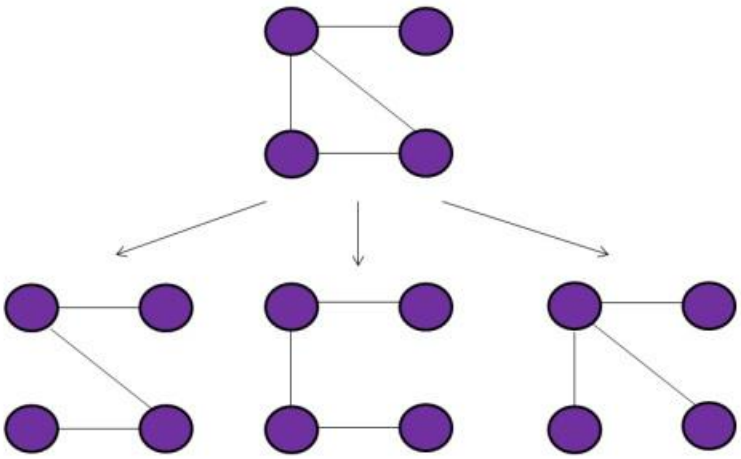
Un algoritmo de aproximación es un algoritmo usado para encontrar soluciones aproximadas a problemas de optimización. Están a menudo asociados con problemas **NP-hard**; como es poco probable que alguna vez se descubran algoritmos eficientes de tiempo polinómico que resuelvan exactamente problemas de este tipo, se opta por encontrar soluciones no-óptimas en tiempo polinomial.

A continuación se presenta uno de estos algoritmos, para esto es necesario conocer el concepto de “árbol de expansión mínima”.

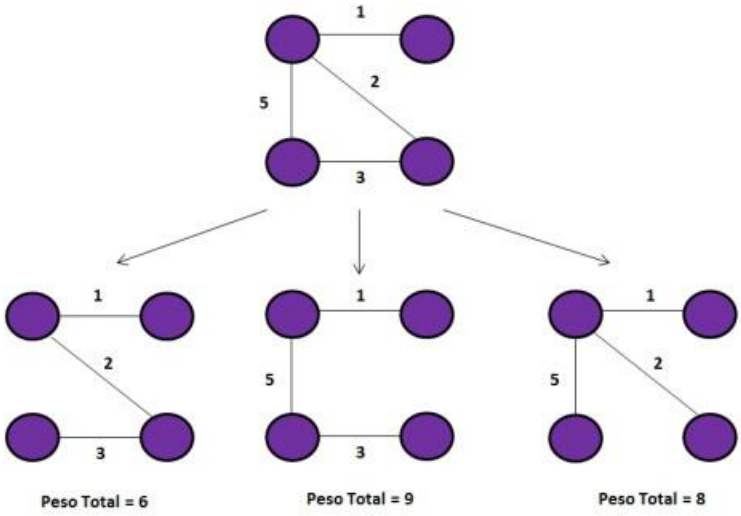
D. Árbol de expansión mínima.

[1]Def. 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, veamos un ejemplo con el siguiente grafo:



[2]Def. 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 sería el primer árbol de expansión cuyo peso total es 6.

Es como tener un conjunto (grafo) y extraer un subconjunto (subgrafo) de este, el cual contiene los mismos nodos del conjunto pero no todas las aristas. Con la condición de que las aristas que tenga sigan conectando a todos los nodos.

El algoritmo principal del que hablaremos es uno de los encargos de encontrar este llamado árbol de expansión mínima, el algoritmo de Kruskal.

E. Algoritmo de Kruskal.

El algoritmo de Kruskal es un algoritmo para encontrar un árbol de expansión mínima en un grafo ponderado. Este algoritmo toma su nombre de Joseph Kruskal, quien lo publicó por primera vez en 1956.

Funciona de la siguiente manera:

- Se crea un grafo P, donde cada vértice del grafo es un árbol separado.
- Se crea un conjunto Q que contenga a todas las aristas del grafo original.
- Mientras Q no esté vació y P no conecte a todos los vértices:
    - Se remueve la arista con peso mínimo de Q.
    - Si la arista que fue removida conecta dos árboles diferentes de F entonces agregamos la arista P, de esta manera dos árboles separados forman uno.

Implementación en Python:

```
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
```

Lo más destacable de este algoritmo es, que al regresar un árbol de expansión mínima, un subgrafo de nuestro grafo original, este cumple con las características que necesitamos. Que pasa por todos los nodos al menos una vez y que la suma de los pesos fuera la mínima.

F. Vecino más cercano. (Heurística)

El Algoritmo del vecino más próximo (***NN*** *por sus siglas en inglés*) o también llamado algoritmo voraz (greedy) permite al viajante elegir la ciudad no visitada más cercana como próximo movimiento. Este algoritmo retorna rápidamente una ruta corta.

Esto es,

- Se toma un vértice origen y se agrega a una lista.
- Se revisa cuáles son sus vecinos, elige el de menor peso revisando que no esté en la lista. Se agrega a la lista.
- Así sucesivamente hasta que se tenga en la lista todos los vértices del grafo.
- Se agrega el vértice origen pero al final de la lista.

Recordemos que parte del problema es regresar a la cuidad de origen.

Es como un ciclo, el cual es conocido como Ciclo Hamiltoniano.

Para N ciudades aleatoriamente distribuidas en un plano, el algoritmo en promedio retorna un camino de un 25% más largo que el menor camino posible

Implementación en Python:

```
def vecinoMasCercano(self):
    lv = list(self.V)
    random.shuffle(lv)
    ni = lv.pop()
    le = dict()
    while len(lv)>0:
        ln = self.v[ni]
        for nv in ln:
            le[nv]=self.E[(ni,nv)]
        menor = min(le.values())
        lv.append(menor)
        del lv[menor]
    return lv
```

G. Solución exacta.

Como mencionamos antes para una solución exacta tenemos que considerar todas las rutas posibles, aquí el algoritmo que realiza estas permutaciones.

```
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
```

Ejercicio. –

Imaginemos que somos unos fervientes seguidores del partido nacionalsocialista. Residimos en Múnich, Alemania y queremos visitar los siguientes 10 lugares vestigios de la Alemania Nazi:

- **Brandeburgo,** en donde se ubica el campo de concentración de Sachsenhausen.
- **Berlín;** museo “Topografía del terror”, donde antiguamente se ubicaba la dirección de las SS.
- **Núremberg,** en 1933 Adolf Hitler declara a Núremberg "Ciudad de los Congresos Partidarios del Tercer Reich". A partir de entonces cada año se reunían durante una semana alrededor de 500.000 nacionalsocialistas de todo el Reich.
- **Baviera,** Campo de concentración de Dachau.
- **Voivodato de Pequeña Polonia,** donde quedan vestigios del más grande campo de concentración Auschwitz. *
- **Rheinlad–Pfalz,** en la iglesia del pueblo Herxheim am Berg se encuentra la campana de Hitler. Es una campana que tiene grabado el nombre de Adolf Hitler y el lema “*Todo por la patria”.*
- **Berchtesgaden,** aquí se encuentra el Kehlsteinhaus, o más conocido como Nido del águila, fue un regalo del partido nazi a Hitler por su 50° cumpleaños.
- **Mauthausen,** lugar donde residen los vestigios del campo de concentración Mauthausen-Gusen.
- **Weimar,** ubicación del campo de concentración de Buchenwald, uno de los más grandes en territorio alemán.

En total tenemos 10 lugares o “ciudades” que queremos visitar, incluyendo Múnich. Múnich es uno de los lugares con más vestigios de la Alemania Nazi, es decir, también nos interesa “visitarlo”.

Bien, entonces necesitamos crear nuestro grafo neonazi.

```
nn=Grafo()
nn.conecta('Múnich','Berlín', 585)

nn.conecta('Múninch','Brandeburgo', 577)
nn.conecta('Berlín','Brandeburgo', 50)

nn.conecta('Múnich','Núremberg', 169)
nn.conecta('Berlín','Núremberg', 437)
nn.conecta('Núremberg','Brandeburgo', 441)

nn.conecta('Múnich','Baviera', 83)
nn.conecta('Baviera','Berlín', 506)
nn.conecta('Baviera','Brandeburgo', 498)
nn.conecta('Baviera','Núremberg', 90)

nn.conecta('Múnich','VPP', 957)
nn.conecta('VPP','Berlín', 654)
nn.conecta('VPP','Brandeburgo', 634)
nn.conecta('VPP','Núremberg', 893)
nn.conecta('VPP','Baviera', 950)

nn.conecta('Múnich','RP', 496)
nn.conecta('RP','Berlín', 666)
nn.conecta('RP','Brandeburgo', 658)
nn.conecta('RP','Núremberg', 351)
nn.conecta('RP','Baviera', 444)
nn.conecta('RP','VPP', 1149)

nn.conecta('Múnich','Berch', 154)
nn.conecta('Berch','Berlín', 736)
nn.conecta('Berch','Brandeburgo', 728)
nn.conecta('Berch','Núremberg', 320)
nn.conecta('Berch','Baviera', 235)
nn.conecta('Berch','VPP', 779)
nn.conecta('Berch','RP', 647)

nn.conecta('Múnich','Mauthausen', 288)
nn.conecta('Mauthausen','Berlín', 597)
nn.conecta('Mauthausen','Brandeburgo', 610)
nn.conecta('Mauthausen','Núremberg', 351)
nn.conecta('Mauthausen','Baviera', 315)
nn.conecta('Mauthausen','VPP', 618)
nn.conecta('Mauthausen','RP', 690)
nn.conecta('Mauthausen','Berch', 173)

nn.conecta('Múnich','Weimar', 395)
nn.conecta('Weimar','Berlín', 285)
nn.conecta('Weimar','Brandeburgo', 278)
nn.conecta('Weimar','Núremberg', 245)
nn.conecta('Weimar','Baviera', 318)
nn.conecta('Weimar','VPP', 768)
nn.conecta('Weimar','RP', 393)
nn.conecta('Weimar','Berch', 553)
nn.conecta('Weimar','Mauthausen', 559)

print(nn.kruskal())
```

Estamos considerando el “peso” como km.
Hay un errorcillo en Múninch debería ser Múnich. Fue corregido al probar los algoritmos.

Camino más corto con:

- Algoritmo de aprox. Kruskal.

```
k=nn.kruskal()
for r in range(10):
    ni=random.choice(list(k.V))
    dfs=k.DFS(ni)
    c=0
    for f in range(len(dfs)-1):
        c+= nn.E[(dfs[f],dfs[f+1])]
        print(dfs[f],dfs[f+1],nn.E[(dfs[f],dfs[f+1])])

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

```

Obtuvimos un árbol de expansión mínima de un “costo” de: 3056
Donde el camino es:

VPP a Mauthausen 618
Mauthausen a Berch 173
Berch a Múnich 154
Múnich a Baviera 83
Baviera a Núremberg 90
Núremberg a Weimar 245
Weimar a Brandeburgo 278
Brandeburgo a Berlín 50
Berlín a RP 666
RP a VPP 1149
costo 3506

- Algoritmo de heurística Vecino más cercano.

Fueron encontradas diversas soluciones pero la mejor fue:

['Brandeburgo', 'Berlín', 'Weimar', 'Núremberg', 'Baviera', 'Múnich', 'Berch', 'Mauthausen', 'VPP', 'RP']

Brandeburgo a Berlín 50 | Berlín a Weimar 28 | Weimar a Núremberg 24 | Núremberg a Baviera 90 | Baviera a Múnich 83 | Múnich a Berch 154 | Berch a Mauthausen 173 | Mauthausen a VPP 618 | VPP a RP 1149 | RP a Brandeburgo 658 |

costo 3505

Hubo también de 3506, que fue el camino que tomo Kruskal.

- Método exacto.

Al utilizar el algoritmo de permutaciones, tardo 1:29.65 min en dar resultados.

Conclusiones. –

Siendo que las ciudades a visitar fueron la mínima cantidad de 10, se intentó con un grafo con más nodos y la diferencia entre los algoritmos fue abismal.

Definitivamente el usar algoritmos de aproximación para este tipo de problemas es lo más factible que se puede hacer. Teniendo un grafo de 50 nodos el algoritmo de permutaciones se tardó alrededor de 2 horas y media en dar resultados, es decir además de esperar todo ese tiempo para solo tener todos los caminos después habría que revisar cada camino.

En particular el Algoritmo Kruskal tiene un gran desempeño en comparación a el Vecino más cercano aunque con este encontré una mejor solución.