Reporte Algoritmo de aproximación

Matemáticas Computacionales

Yarethzi Giselle Bazaldúa Parga

06 de noviembre de 2017

Resumen

Los algoritmos de aproximación están siendo cada vez más utilizados para resolver problemas donde los algoritmos exactos de tiempo polinomial son conocidos pero demasiado costosos debido al tamaño de la entrada.

En el Problema del Agente Viajero(PAV) el objetivo es encontrar un recorrido completo que conecte todos los nodos de una red, visitándolos tan solo una vez y volviendo al punto de partida, y que además minimice la distancia total de la ruta.

Este tipo de problemas tiene gran aplicación en el ámbito de la logística y distribución, así como en la programación de curvas de producción.

La complejidad del cálculo del problema del agente viajero ha despertado múltiples iniciativas por mejorar la eficiencia en el cálculo de rutas. El método más básico es el conocido con el nombre de fuerza bruta, que consiste en el cálculo de todos los posibles recorridos, lo cual se hace extremadamente ineficiente y casi que se imposibilita en redes de gran tamaño. También existen heurísticos que se han desarrollado por la complejidad en el cálculo de soluciones óptimas en redes robustas, es por ello que existen métodos como el

vecino más cercano, la inserción más barata y el doble sentido. Por último se encuentran los algoritmos que proporcionan soluciones óptimas, como el método de branch and bound (ramificación y poda), que trabaja el problema como un algoritmo de asignación y lo resuelve por medio del método simplex.

Un algoritmo de aproximación es un algoritmo usado para encontrar soluciones aproximadas a problemas de optimización. Más aun, es frecuente que estos algoritmos posean un desempeño práctico muy superior a su garantía teórica

El algoritmo de Kruskal 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 de la suma de todas las aristas del árbol es el mínimo. Primeramente ordenaremos las aristas del grafo por su peso de menor a mayor. Como hemos ordenado las aristas por peso comenzaremos con la arista de menor peso, si los vértices que contienen dicha arista no están en la misma componente conexa entonces los unimos para formar una sola componente mediante Union(x , y) al hacer esto estamos evitando que se creen ciclos y que la arista que une dos vértices siempre sea la mínima posible.

Dado un grafo conexo, 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.

El problema que se presenta a resolver por el método del algoritmo Kruskal es el siguiente:

Una empresa de piscinas de venta en línea desea repartir sus pedidos en diferentes puntos de la República Mexicana, los destinos son los siguientes:

Monterrey: ‘M’

Ciudad de México: ‘CM’

Guadalajara: ‘G’

Veracruz: ‘V’

Chihuahua: ‘C’

Saltillo: ‘S’

Acapulco: ‘A’

Mazatlán: ‘MZ’

Zacatecas: ‘Z’

Morelia: ‘MO’

Y las distancias (en kilómetros) entre ellos se muestran a continuación:

Distancia	Mty	Cd. De México	Guadalajara	Veracruz	Chihuahua	Saltillo	Acapulco	Mazatlán	Zacatecas	Morelia
Mty	---	913	794	1229	803	87	1286	838	462	828
Cd. De México	913	---	536	419	369	844	379	1021	601	299
Guadalajara	794	536	---	903	1167	701	901	490	341	288
Veracruz	1229	419	903	---	1781	1197	731	1388	954	665

Chihuahua	803	369	1167	1781	---	726	1814	936	832	1260
Saltillo	87	844	701	1197	726	---	1219	757	377	762
Acapulco	1286	379	901	731	1814	1219	---	1386	974	663
Mazatlán	838	1021	490	1388	936	757	1386	---	543	767
Zacatecas	462	601	341	954	832	377	974	543	---	437
Morelia	828	299	288	665	1260	762	663	767	437	---

En el siguiente algoritmo de Kruskal se aplican los datos mostrados:

```
>>> from copy import deepcopy
>>> import random
>>> 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
        class Fila:
                def __init__(self):
                        self.fila= []
                def obtener(self):
                        return self.fila.pop()
                def meter(self,e):
                        self.fila.insert(0,e)
                        return len(self.fila)
                @property
                def longitud(self):
                        return len(self.fila)
        class Pila:
                def __init__(self):
                        self.pila= []
                def obtener(self):
                        return self.pila.pop()
                def meter(self,e):
                        self.pila.append(e)
```

```
                return len(self.pila)
        @property
        def longitud(self):
                return len(self.pila)
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 BFS(self,ni):
                visitados =[]
                f=Fila()
```

```
                f.meter(ni)
                while(f.longitud>0):
                        na = f.obtener()
                        visitados.append(na)
                        ln = self.vecinos[na]
                        for nodo in ln:
                                if nodo not in visitados:
                                        f.meter(nodo)
                return visitados
        def DFS(self,ni):
                visitados =[]
                f=Pila()
                f.meter(ni)
                while(f.longitud>0):
                        na = f.obtener()
                        visitados.append(na)
                        ln = self.vecinos[na]
                        for nodo in ln:
                                if nodo not in visitados:
                                        f.meter(nodo)
                return visitados
        def shortest(self, v): #Dijkstra´s algorithm
                q = [(0, v, ())] #arreglo "q" de las "Tuplas" de lo que se va a almacenar donde 0 es la distancia, v el nodo y () e
                dist = dict() #diccionario de distancias
                visited = set() #conjunto de visitados
                while len(q) > 0: #mientras exista un nodo pendiente
                        (l, u, p) = heappop(q) #se toma la tupla con la distancia menor
                        if u not in visited: #si no lo hemos visitado
                                visited.add(u) #se agrega a visitado
                                dist[u] = (l,u,list(flatten(p))[::-1] + [u]) #agrega al diccionario
                        p = (u, p) #Tupla del nodo y el camino
                        for n in self.vecino[u]: #Para cada hijo del nodo actual
                                if n not in visited: #si no lo hemos visitado
```

```
                                        el = self.E[(u,n)] #se toma la distancia del nodo actual hacia el nodo hijo
                                        heappush(q, (l + el, n, p)) #se agrega al arreglo "q" la distancia actual mas la distancia
                return dist #regresa el diccionario de distancias
        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
        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
g= Grafo()
g.conecta('M','CM', 913)
g.conecta('M','G', 794)
g.conecta('M','V', 1229)
g.conecta('M','C', 803)
g.conecta('M','S', 87)
g.conecta('M','A', 1286)
g.conecta('M','MZ', 838)
g.conecta('M','Z', 462)
g.conecta('M','MO', 828)
g.conecta('CM','G', 536)
g.conecta('CM','V', 419)
g.conecta('CM','C', 369)
g.conecta('CM','S', 844)
g.conecta('CM','A', 379)
g.conecta('CM','MZ', 1021)
g.conecta('CM','Z', 601)
g.conecta('CM','MO', 299)
g.conecta('G','V', 903)
g.conecta('G','C', 1167)
g.conecta('G','S', 701)
g.conecta('G','A', 901)
g.conecta('G','MZ', 490)
g.conecta('G','Z', 341)
g.conecta('G','MO', 288)
g.conecta('V','C', 1781)
g.conecta('V','S', 1197)
```

```
        g.conecta('V','A', 731)
        g.conecta('V','MZ', 1388)
        g.conecta('V','Z', 954)
        g.conecta('V','MO', 665)
        g.conecta('C','S', 726)
        g.conecta('C','A', 1814)
        g.conecta('C','MZ', 936)
        g.conecta('C','Z', 832)
        g.conecta('C','MO', 1260)
        g.conecta('S','A', 1219)
        g.conecta('S','MZ', 757)
        g.conecta('S','Z', 377)
        g.conecta('S','MO', 762)
        g.conecta('A','MZ', 1386)
        g.conecta('A','Z', 974)
        g.conecta('A','MO', 663)
        g.conecta('MZ','Z', 543)
        g.conecta('MZ','MO', 767)
        g.conecta('Z','MO', 437)
        print(g.kruskal())
        #print(g.shortest('c'))
        print(g)
        k = g.kruskal()
        print([print(x, k.E[x]) for x in k.E])
        for r in range(10):
                ni = random.choice(list(k.V))
                dfs = k.DFS(ni)
                c = 0
                #print(dfs)
                #print(len(dffs))
                for f in range(len(dfs) -1):
                        c += g.E[(dfs[f],dfs[f+1])]
                        print(dfs[f], dfs[f+1], g.E[(dfs[f],dfs[f+1])])


                 c += g.E[(dfs[-1],dfs[0])]
                 print(dfs[-1], dfs[0], g.E[(dfs[-1],dfs[0])])
                 print('costo',c)
        dfs = g.vecinoMasCercano()
        print(dfs)
        c=0
        for f in range(len(dfs) -1):
                c += g.E[(dfs[f],dfs[f+1])]
                print(dfs[f], dfs[f+1], g.E[(dfs[f],dfs[f+1])])
        c += g.E[(dfs[-1],dfs[0])]
        print(dfs[-1], dfs[0], g.E[(dfs[-1],dfs[0])])
        print('costo',c)
        data = list('MCMGVCSAMZZMO')
        #data = ['mty','saltillo','chi']
        tim=time.clock()
        per = permutation(data)
        print(time.clock()-tim)
```