REPORTE DEL ALGORTIMO DISJKTRA

Elaborado por: Luis Daniel Honorato Hernández

Licenciatura en Matemáticas

1729448

INTRODUCCION

En este presente reporte se abordará el problema del agente viajero en el cual se verá algunos conceptos fundamentales y métodos de solución que se deben de apreciar para poder realizar correctamente el algoritmo y después se trabajara ya con estos conocimientos en el algoritmo en si en encontrar las mínimas distancias en tiempo de un listado de varias ciudades seleccionadas y al final se realizaran conclusiones acerca de lo aprendido en este reporte

PROBLEMA DEL AGENTE VIAJERO

El problema del agente viajero se define como el hallar un recorrido completo que cumpla con conectar todos los vértices de un grafo visitándolos solamente una vez y al final se debe de regresar hacia el punto donde se dio comienzo y al igual en todo ese recorrido hecho se busca hacer la mínima distancia total recorrida

Además de que este problema tiene una variación importante y está condicionada en las distancias que existen de un vértice a otro al decir que pueden que sean simétricas o no en otras palabras que la distancia que existe entre el vértice A al vértice B sea igual a la distancia del vértice B al vértice A puesto que cuando se realice en un programa sea la mínima posibilidad de que así sea interpretado y para ayudarnos a obtener esas distancias para calcular después las rutas posibles que existen se encuentra una ecuación en la cual el número de vértices que tu tengas restándolo uno todo en un factorial es decir (n-1)! , y solo se elimina una posible ruta para no volver a repetir ese mismo y ahórrate espacio y tiempo dará como resultado como se mencionó anteriormente la cantidad de rutas posibles y es por eso que el problema podría presentar una gran complejidad por el número de vértices que tenga el programa a desarrollar pero en caso de que sea el problema simétrico como se explicó previamente la cantidad de rutas posibles se reduce solamente a la mitad es decir((n-1)!)/2 lo cual nos ayuda a un ahorro del tiempo y espacio del programa a realizar

ALGORITMO DE APROXIMACION

A veces un algoritmo como los realizados en reportes anteriores en el momento de que se vera la solución en pantalla esta tiene un tiempo de duración muy demorante y es por eso por lo que en estos posibles casos se usa el algoritmo de aproximación el cual busca aproximar la solución, pero no será igual a la solución original

ARBOL DE EXPANSION MINIMA

El árbol de expansión mínima se podría definir como un modelo de optimización en el cual todos los vértices del árbol se deben de enlazar ya sea en forma directa o indirecta con el principal objetivo de que al hacer la suma total de todos los pesos de los vértices sea la mínima posible y esto se lograra comenzando con el arco de menor longitud después de este se seleccionara los caminos posibles que enlazan a este mínimo seleccionando un arco mínimo y así sucesivamente hasta que en el algoritmo se encuentren todos los arcos conectados

HEURISTICA DE VECINO MAS CERCANO

La heurística del vecino más cercano es un método que sirve para solucionar el problema del agente viajero pero este no logra asegurarnos al 100% si la solución encontrada es la más optima a nuestro problema sin embargo suele proporcionar buenas soluciones y su tiempo de cálculo del procedimiento es el más eficiente y su procedimiento es similar a cuando se obtiene el árbol de expansión mínima donde establecido el nodo inicial se parte a encontrar el vecino más cercano a este es decir que se encuentre en su ramificación y que cumpla que el peso de su arista sea el menor posible y después se parte de esa ramificación para encontrar el siguiente y así sucesivamente hasta llegar al nodo de partida

SOLUCION EXACTA

Este método que se utiliza en problemas de optimización como cuando se trata de buscar la mejor ruta optima de entre varias ciudades o cualquier lugar al que se desee llegar tomando en cuenta factores como el tiempo y sobre todo lo más importante tomar las rutas con menor distancia para realizar más rápido y eficaces las cosas nos sirve usar la solución exacta ya que su principal objetivo es darnos la solución exacta de todas las posibles combinaciones que se genera del número de ciudades o elementos a recorrer y al igual debe de revisar de entre todas las combinaciones cuales es la más eficaz es decir la que tiene menor peso

PROGRAMA DE AGENTE VIAJERO

El siguiente código a mostrar a continuación tiene como objetivo como se mencionó previamente encontrar el mínimo trayecto en este caso será la mínima distancia entre las 10 principales ciudades de Estados Unidos donde se comparara tanto con el árbol de expansión mínima, la heurística del vecino más cercano y la solución exacta los resultados que arrojen la mínima distancia recorrida a base del costo y tiempo total y al final se verán los resultados en una tabla donde se podrá visualizar mejor las comparaciones ya que antes de ello se deberán de ver todos los resultados mostrados en cada ejecución del programa para ir comparando

Las ciudades por evaluar son:

a.- Los Ángeles

b.- San Francisco

c.- Nueva York

d.- Chicago

e.- Miami

f.- Baltimore

g.- San Antonio

h.- Austin

i.- Phoenix

j.- Indianápolis

#CODIGO A MOSTRAR

```
from heapq import heappop,heappush
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 shortests(self,v):#algoritmo de dijkstra
        q=[(0,v,())]#arreglo q de las tuplas de lo que se va a almacenar donde 0 es la distancia,
v el nodo y() el camino hacia el
        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 visitados
                dist[u]=(l,u,list(flatten(p))[::-1]+[u])#agrega el diccionario
            p=(u,p)#tupla del nodo y el camino
```

```
            for n in self.vecinos[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 mas la distancia hacia el nodo hijo
                    heappush(q,(l+el,n,p))#se agrega el arreglo q la distancia actual mas la distancia hacia el nodo hijo n hacia donde se va y el camino
        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('a','b', 381)
g.conecta('a','c', 2789)
g.conecta('a','d', 2015)
g.conecta('a','e', 2733)
g.conecta('a','f', 2655)
g.conecta('a','g', 1352)
g.conecta('a','h', 1377)
g.conecta('a','i', 373)
g.conecta('a','j', 2071)
g.conecta('b','c', 2905)
g.conecta('b','d', 2131)
g.conecta('b','e', 3113)
g.conecta('b','f', 2818)
g.conecta('b','g', 1733)
```

```
g.conecta('b','h', 1758)
g.conecta('b','i', 753)
g.conecta('b','j', 2275)
g.conecta('c','d', 789)
g.conecta('c','e', 1284)
g.conecta('c','f', 192)
g.conecta('c','g', 1823)
g.conecta('c','h', 1743)
g.conecta('c','i', 2408)
g.conecta('c','j', 709)
g.conecta('d','e', 1377)
g.conecta('d','f', 702)
g.conecta('d','g', 1240)
g.conecta('d','h', 1161)
g.conecta('d','i', 1753)
g.conecta('d','j', 181)
g.conecta('e','f', 1098)
g.conecta('e','g', 1383)
g.conecta('e','h', 1352)
g.conecta('e','i', 2360)
g.conecta('e','j', 1197)
g.conecta('f','g', 1640)
g.conecta('f','h', 1560)
g.conecta('f','i', 2293)
g.conecta('f','j', 594)
g.conecta('g','h', 79)
g.conecta('g','i', 981)
g.conecta('g','j', 1172)
g.conecta('h','i', 1066)
g.conecta('h','j', 1094)
```

```
g.conecta('i','j', 1703)



print(g.kruskal())
#print(g.shortests('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(dfs))
    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)

vmc = g.vecinoMasCercano()
print(vmc)
c=0
for f in range(len(vmc) -1):
    c += g.E[(vmc[f],vmc[f+1])]
    print(vmc[f], vmc[f+1], g.E[(vmc[f],vmc[f+1])] )
```

```
c += g.E[(vmc[-1],vmc[0])]
print(vmc[-1], vmc[0], g.E[(vmc[-1],vmc[0])])
print('vmc costo',c)



data=list('abcdefghij')
tim=time.clock()
per=permutation(data)
vm, rm= 100000000000,[]
for e in per:
    #print(e)
    c=0
    for f in range(len(e) -1):
        c += g.E[(e[f],e[f+1])]
        #print(e[f], e[f+1], g.E[(e[f],e[f+1])] )

    c += g.E[(e[-1],e[0])]
    #print(e[-1], e[0], g.E[(e[-1],e[0])])
    if c < vm:
        vm,rm= c,e
    #print('e costo',c)
print(time.clock()-tim)
print('minimo exacto',vm,rm)
```

<u>RESULTADOS CON SUS COMPARACIONES</u>

ARBOL DE EXPANSION MINIMA	HEURISTICA DEL VECINO MAS CORTO	SOLUCION EXACTA
b a 381 a i 373 i g 981 g h 79	['j', 'd', 'f', 'c', 'e', 'h', 'g', 'i', 'a', 'b'] j d 181 d f 702	72.97390175652546 mínimo exacto 7477 ['a', 'b', 'd', 'j', 'c', 'f', 'e', 'h', 'g', 'i']

h j 1094 j f 594 f c 192 c e 1284 e d 1377 d b 2131 costo 8486 c f 192 f j 594 j h 1094 h g 79 g i 981 i a 373 a b 381 b d 2131 d e 1377 e c 1284 costo 8486 b a 381 a i 373 i g 981 g h 79 h j 1094 j f 594 f c 192 c e 1284 e d 1377 d b 2131 costo 8486 72.97390175652546	f c 192 c e 1284 e h 1352 h g 79 g i 981 i a 373 a b 381 b j 2275 vmc costo 7800 72.97390175652546	
c f 192 f e 1098 e j 1197 j h 1094 h g 79 g i 981 i a 373 a b 381 b d 2131 d c 789 costo 8315 71.04999652860579	['e', 'f', 'c', 'j', 'd', 'h', 'g', 'i', 'a', 'b'] e f 1098 f c 192 c j 709 j d 181 d h 1161 h g 79 g i 981 i a 373 a b 381 b e 3113 vmc costo 8268 71.04999652860579	71.04999652860579 mínimo exacto 7477 ['a', 'b', 'd', 'j', 'c', 'f', 'e', 'h', 'g', 'i']
d j 181 j f 594 f c 192 c e 1284 e h 1352 h g 79 g i 981	['i', 'a', 'b', 'g', 'h', 'j', 'd', 'f', 'c', 'e'] i a 373 a b 381 b g 1733 g h 79 h j 1094	69.89774238632694 mínimo exacto 7477 ['a', 'b', 'd', 'j', 'c', 'f', 'e', 'h', 'g', 'i']

i a 373 a b 381 b d 2131 costo 7548 j f 594 f c 192 c e 1284 e h 1352 h g 79 g i 981 i a 373 a b 381 b d 2131 d j 181 costo 7548 69.89774238632694	j d 181 d f 702 f c 192 c e 1284 e i 2360 vmc costo 8379 69.89774238632694	
j f 594 f c 192 c e 1284 e h 1352 h g 79 g i 981 i a 373 a b 381 b d 2131 d j 181 costo 7548 d j 181 j f 594 f c 192 c e 1284 e h 1352 h g 79 g i 981 i a 373 a b 381 b d 2131 costo 7548 76.58000054334865	['j', 'd', 'f', 'c', 'e', 'h', 'g', 'i', 'a', 'b'] j d 181 d f 702 f c 192 c e 1284 e h 1352 h g 79 g i 981 i a 373 a b 381 b j 2275 vmc costo 7800 76.58000054334865	76.58000054334865 mínimo exacto 7477 ['a', 'b', 'd', 'j', 'c', 'f', 'e', 'h', 'g', 'i']
f j 594 j d 181 d h 1161 h g 79 g i 981 i a 373 a b 381 b e 3113 e c 1284	['a', 'i', 'b', 'g', 'h', 'j', 'd', 'f', 'c', 'e'] a i 373 i b 753 b g 1733 g h 79 h j 1094 j d 181 d f 702	70.99341340682626 mínimo exacto 7477 ['a', 'b', 'd', 'j', 'c', 'f', 'e', 'h', 'g', 'i']

c f 192 costo 8339 f j 594 j d 181 d h 1161 h g 79 g i 981 i a 373 a b 381 b e 3113 e c 1284 c f 192 costo 8339 70.99341340682626	f c 192 c e 1284 e a 2733 vmc costo 9124 70.99341340682626	

CONCLUSIONES

Como se puede observar en los resultados mostrados en la tabla se puede notar que hay costos iguales en varios intentos pero iniciando en distintos nodos por el método del árbol de expansión mínima que al compararse con la heurística del vecino más cercano tiene una gran diferencia ya que o se cumple que el árbol de expansión mínima es menor al vecino más cercano o viceversa y al compararse al final con la solución exacta esta resulta ganadora porque es el más eficaz pero en realidad al momento de programar los más eficaces son el árbol de expansión mínima o la heurística del vecino más corto que esto queda a disposición de cada persona ya que se tardan menos en ejecutar que la solución exacta ya que se tarda un poco más de tiempo y además no se garantiza que sea la solución exacta a pesar de ser la misma en cada ejecución y con respecto a los tiempos de cada ejecución fueron decrementando e incrementando constantemente así que se demuestra al final que ningún método es mejor que otro porque esto dependerá del uso al que se le esté aplicando en cualquiera de los casos que se presente en la vida diaria.