Universidad de los Andes

FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS
DISEÑO Y ANÁLISIS DE ALGORITMOS

TAREA 5 - PARTE 4

Sergio Pardo Gutierrez
Juan Diego Calixto
Juan Diego Yepes

19 de Abril de 2022
Bogotá D.C.

Tabla de contenidos

1 Enunciado

Una ciudad se diseño de tal modo que todas sus calles fueran de una sola vía. Con el paso del tiempo la cantidad de habitantes de la ciudad creció y esto produjo grandes trancones en algunas de las vias debido a algunos desvíos innecesarios que tienen que tomar los habitantes de la ciudad para poder llegar a sus trabajos. Por lo tanto, el alcalde tomó la decisión de ampliar algunas vias para que puedan convertirse en doble via. Dado el mapa de la ciudad y el costo de convertir cada via actual en doble via, determinar qué vias se deben convertir, de modo que se pueda transitar de cualquier punto a cualquier punto de la ciudad por dobles vias y que el costo de la conversión sea el mínimo posible.

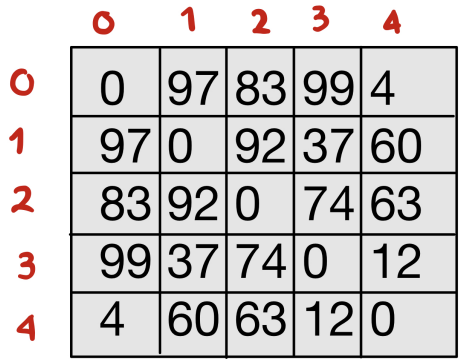
2 Solución

2.1 Entradas y salidas

Para resolver este problema primero se detallan las entradas y salidas:

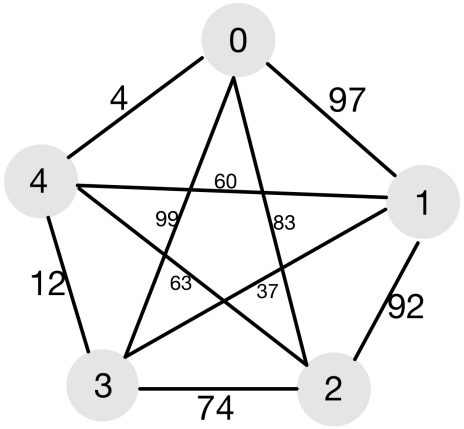
E/S	Nombre	Tipo	Significado
E	matrix	`list`	Matriz con los pesos de construir cada vía
S	mstInfo	`dict`	Diccionario que representa el grafo
S	needed	`list`	Lista con los identificadores de los arcos del `MST`

La entrada será una matriz de adyacencias que se muestra en la imagen.

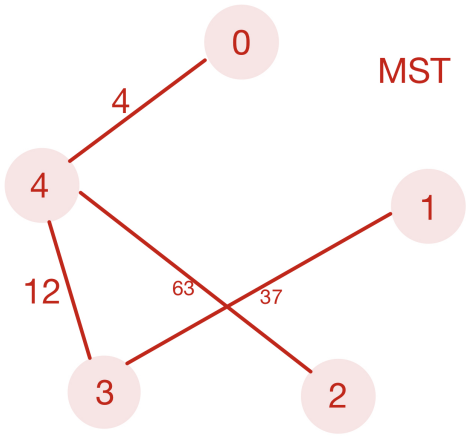


Ésta matriz debe ser simétrica para dar un resultado consistente. Esta decisión fue tomada ya que la lógica del problema lo permite, puesto que solo necesitábamos saber los costos de construir la vía doble. Claramente esto indica que nuestro grafo no es dirigido, que los vértices representan las intersecciones entre las vías y los el costo de construir las vías.

La matriz daría lugar al siguiente grafo:



Por otro lado, las salidas forman un árbol de mínimo recubrimiento del grafo pero expresado como un diccionario, que se debe ver de la siguiente forma:



La primera salida es una lista con los arcos que funcionan, `[4,12,37,63]` y un diccionario con parejas llave valor de la forma

```
{
    3: [4, 0, 4], # Las parejas llave valor representan lo siguiente
    9: [12, 3, 4], # La llave es el identificador del arco
    5: [37, 1, 3], # El primer elemento de la lista del valor es el costo del arco
    6: [60, 1, 4], # El segundo elemento de la lista del valor es v rtice de inicio
    8: [63, 2, 4], # El tercer elemento de la lista del valor es v rtice de llegada
    7: [74, 2, 3],
    1: [83, 0, 2],
    4: [92, 1, 2],
    0: [97, 0, 1],
    2: [99, 0, 3]
}
```

Como estamos trabajando con entradas y salidas arbitrarias, buscamos que fueran lo más amigables posible con el usuario luego para cada arco del grafo se imprime una cadena de caracteres con el siguiente formato: "Se necesita la autopista de costo x que va desde el nodo n hasta el nodo m". Para esto se buscan en el diccionario de retorno los valores de las llaves identificadores de la lista de retorno.

2.2 Algoritmo

Con base en esta aproximación al problema, es evidente que seleccionamos el algoritmo de Kruskal que genera un árbol de recubrimiento mínimo sobre el grafo. Como se especificó antes, el algoritmo retorna una lista con los ejes de menor costo tal que formen una estructura que conecte todos los nodos

Al implementar el algoritmo desarrollado con la matriz de ejemplo el programa genera el siguiente resultado:

```
Se necesita la doble vía de costo 4 que va desde el nodo 0 hasta el nodo 4
Se necesita la doble vía de costo 12 que va desde el nodo 3 hasta el nodo 4
Se necesita la doble vía de costo 37 que va desde el nodo 1 hasta el nodo 3
Se necesita la doble vía de costo 63 que va desde el nodo 2 hasta el nodo 4
```

2.3 Explicación

El algoritmo funciona creando un diccionario que guarda los ejes con un como llave id y guardando como valor el costo de la doble vía, su lugar de origen y destino. Además se crea una lista que guarda todos los ids de los ejes. Después, se ordenan los ids de los ejes con base en el peso del eje que representan. Posteriormente se itera sobre cada eje en esta lista y se verifica si crea un ciclo o no en el grafo. Si no lo crea, el eje es agregado a la lista que guarda el árbol de recubrimiento mínimo. Si crea un ciclo simplemente continúa el proceso con el siguiente eje.

Para detectar un ciclo se utiliza la estrategia de los representantes de cada partición y se aplana el árbol a través de esta estructura para reducir la complejidad temporal.

2.4 Casos de prueba

Para probar el algoritmo de Kruskal hay 3 archivos de prueba identificados con el sufijo kruskal. Hay un caso de un grafo de 5 vértices, uno de 100 y uno de 1000. Todos estos completamente conectados como el mostrado en el ejemplo. Sin embargo, el algoritmo puede probarse con cualquier otro archivo de texto y dará una respuesta con sentido, pero semánticamente incorrecta puesto que estos archivos están diseñados para grafos dirigidos y el algoritmo ignorará la mitad de la matriz. Si desea generar un caso de prueba nuevo, puede ir ejecutar el siguiente comando en la terminal

```
Python graphKruskal.py > distances100_kruskal.txt
```

cambiando el numero del archivo de texto (este es solo indicativo). Para que se genere el grafo con el tamaño deseado, solo debe ir a la linea 32 del archivo `graphKruskal.py` y cambiar el 0 por el valor deseado. Para correr el programa en terminal utilice el comando

```
Python kruskal.py < distances5_.txt
```

y cambie el nombre del archivo por el que desea ejecutar. Puede agregar también un archivo de salida para guardar el resultado:

```
Python kruskal.py < distances5_kruskal.py > respuesta.txt
```

3 Código fuente

```
import sys
import math
import time

def kruskal(matrix:list):
    roads={}
    i,j,k=0,0,0
    while i < len(matrix):
        j=i
        while j < len(matrix):
            cost = matrix[i][j]
            if cost != 0 and cost != -1:
                roads[k]=[cost,i,j]
                k+=1
            j+=1
        i+=1

    roads = dict(sorted(list(roads.items()),key=lambda x:x[1][0]))
    partition = createPartition(len(matrix))
    roadsNeeded=[]
    for i in roads:
        start = roads[i][1]
        end = roads[i][2]
        if not sameSubset(start, end, partition):
            roadsNeeded.append(i)
            union(start, end, partition)
    print(roads)
    return roadsNeeded, roads

def createPartition(sizeV:int)->dict:
    partition = {}
    i=0
    while i < sizeV:
        partition[i]={"parent":i,"height":1}
        i+=1
    return partition

def find(v:int, partition:dict)->int:
    if partition[v]["parent"]==v: #es su mismo representante (caso base)
        return v
    s = find(partition[v]["parent"],partition) #llamado recursivo, subimos en la
    recursi n para saber qui n es la ra z
    partition[v]["parent"] = s #aplanar arbol
    return s


```

```
46 def sameSubset(v1:int,v2:int, partition: dict)->bool:
47     return find(v1,partition)==find(v2,partition)
48 
49 def union(v1:int, v2:int, partition)->None:
50     s1 = find(v1,partition)
51     s2 = find(v2,partition)
52     if (partition[s1]["height"]<partition[s2]["height"]): #si el arbol tiene mayor
       altura
53         partition[s1]["parent"]=s2; # pasarle el representante a apuntar al conjunto
       con el que queremos unirlo
54     else:
55         partition[s2]["parent"]=s1
56         if (partition[s1]["height"] == partition[s2]["height"]):
57             partition[s2]["height"] += 1; #solo se actualiza la altura del
       representante. (est n mal calculados pero es suficiente)
58 
59 matrix = []
60 lines=sys.stdin.readlines()
61 c = 0
62 while c<len(lines)and lines[c]!="\n":
63     line = lines[c].strip().replace("\n","").split("\t")
64     lineInt = []
65     for value in line:
66         valInt = int(value)
67         lineInt.append(valInt)
68     matrix.append(lineInt)
69     c+=1
70 answer = kruskal(matrix)
71 roadsNeeded = answer[0]
72 roads = answer[1]
73 for road in roadsNeeded:
74     cost = roads[road][0]
75     origin = roads[road][1]
76     destiny = roads[road][2]
77     print("Se necesita la doble v a de costo {} que va desde el nodo {} hasta el nodo
        {} ".format(cost,origin,destiny))
```