Teoría de algoritmos 1 - 75.29

Trabajo Práctico N°: 1

Integrantes:

|  |  |  |
| --- | --- | --- |
| Padrón | Nombre y Apellido | Email |
| 90009 | Lautaro Medrano | lautaromedrano@hotmail.com |
| 92216 | Alejandro Pernín | ale.pernin@gmail.com |

|  |
| --- |
| Para us ode la cátedra |
| Primera entrega |
| Corrector |
| Observaciones |
|  |
| Segunda entrega |
| Corrector |
| Observaciones |

Indice

[**Enunciado** 3](#_Toc369682290)

[**Estrategia de resolución** 5](#_Toc369682291)

[**Análisis de orden** 7](#_Toc369682292)

[**Compilación y ejecución** 9](#_Toc369682293)

[**Conclusiones** 10](#_Toc369682294)

[**Código fuente** 11](#_Toc369682295)

# **Enunciado**

**Robustez en grafos**

**Definición:** Sea el grado de robustez de un grafo la cantidad mínima d earistas que es necesario remover del grafo para que el mismo sea no conexo.

Desarrollar un algoritmo que dado u grafo y un grado de robustez, enumere todas las aristas que sería necesario agregar al grafo en cuestión, para que el mismo alcance el grado de robustez especificado. Dicho algoritmo debe ser lo más óptimo posible.

Implementar el algoritmo en una aplicación quetome como entrada dos parámetros: el grado de robustez y el nombre del archivo con la definición del grafo.

El archivo con la definición del grafo debe respetar el siguiente formato:

*<vértice>:<verticeadyacente1>,<verticeadyacente2>*

Ejemplo de invocación de la aplicación:

*tdatp1 3 grafo1.txt*

En cuanto a la salida de la aplicación, la misma consistirá en listar por pantalla las aristas a agregar para alcanzar el grado de robustez requerido. El formato de salida es elsiguiente:

*<arista#>:<verticeorigen>,<verticedestino>*

**Ejemplo:**



**B F**

**A D E H**

**C**

**G**

Grafo1:gradoderobustez1

*grafo1*.txt

*A:B,C*

*B:A,D*

*C: A, D D:B,C,E*



*E:D,F,G F:E,H*

*G:E,H*

*H:F,G*

**B F**

**A D E H**

**C**

**G**

Grafo1 modificado: grado de robustez 3

*Salida*

*arista1:A,H arista2:B,F  
arista3:C,G*

# **Estrategia de resolución**

Para resolver el problema planteado se tuvo en cuenta las siguientes hipótesis:

* Todos los grafos ingresados no dirigidos, aún si el archivo del grafo lo dispone así así, el algoritmo automáticamente establece las aristas tanto de A>B como de B>A
* No se admiten aristas paralelas ni bucles.
* Sea un grafo de n vértices, la mayor robustez posible es n – 1.

Si esta última condición no se cumple, no es posible cumplir la robustez solicitada, ya que un grafo de n vértices, completo, tiene a lo sumo n - 1 aristas incidentes en cada vértice, con lo cual, al eliminar n - 1 aristas, el grafo deja de ser conexo.

Para que el grafo posea una robustez ‘x’, cada vértice al menos debe tener ‘x’ aristas incidentes; así también como si se divide el grafo en componentes, cada componente debe tener ‘x’ aritas incidentes.

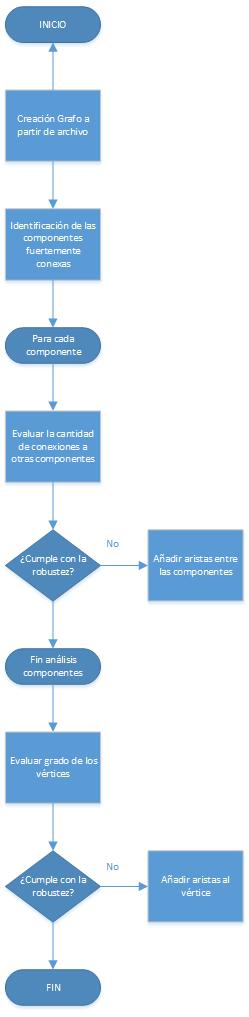
Luego se siguió el siguiente esquema de procesamiento:

* Se identifican las componentes fuertemente conexas del grafo[[1]](#footnote-2).
* Se evalúa la cantidad de conexiones entre las distintas componentes.
* Si alguna componente no cumple con la robustez solicitada se añaden las aristas necesarias para cumplir dicha condición.
* Se analiza el grado de cada vértice para verificar que coincida con la robustez solicitada.

Para identificar las componentes fuertemente conexas del grafo se implementó un DFS (Depth First Search) modificado:

Se almacenan los vértices en una lista hasta que se genera un ciclo, en ese momento, se crea una nueva componente fuertemente conexa y se añade a la lista de las mismas.

Diagrama de flujo del algoritmo (a alto nivel) para obtener el grafo con la robustez solicitada:



# **Análisis de orden**

A continuación se indica el orden de cada una de las primitivas del TDA Grafo, con una justificación del valor del mismo en los casos que sea necesario:

(por cuestiones de simplicidad se omiten aquellos métodos de tipo get)

* countVertices: O (1). Implica una simple llamada en Python para verificar la longitud de un diccionario, para lo cual asumimos que se mantiene una variable que contiene dicha longitud.
* isVertex: O (1). Implica validar si existe una clave en el diccionario de vértices.
* addVertex: O (1). Implica añadir un valor al diccionario de vértices.
* addAdjacency: O (1). Implica validar si ya existe la arista en el diccionario y añadirla en caso contrario. Ambas situaciones se resuelven en O (1).
* isEdge: O (1). Implica validar si el diccionario de adyacencias del vértice de inicio contiene el vértice final. Implica 2 accesos que se resuelven en O (1).
* addEdge: O (1). Implica añadir 2 vértices a 2 diccionarios de adyacencia distintos. Esto se debe a que, al ser el grafo no dirigido, cada arista debe representar los caminos en ambas direcciones. Cada una de estas operaciones se resuelve en O (1).
* rdfs: O (V+E). El tiempo normal de ejecución de DFS es O (V+E).V:Vértices, E:Aristas.('Introduction to Algorithms' Second Edition - Thomas Cormen page 543).

Al agregar un ciclo sobre la lista 'l', sólo se agrega un elemento cuando un vértice está siendo visitado, cada llamada al ciclo va a limpiar la lista y las operaciones dentro del ciclo son de O (1), por lo que en total este ciclo agregará O (V).

En resumen:

O(V+E) + O(V) = O(V+V+E) = O(2V+E) = O(2V) + O(E) = 2O(V) + O(E)

= O(V) + O(E) = O(V+E)

[O(kg)=O(g) si k es constante y distinto de 0]

* addEdgesSets: O (V). Implica, en el peor caso, iterar sobre la lista de vértices del grafo.
* addEdgesVertex: O (V). Implica iterar sobre la lista de vértices del grafo.
* edgesInSets: O (V2). Implica iterar sobre la lista de vértices del Set 2, 1 vez para cada vértice existente en el Set 1.
* setStrenght: O (V2). Toda la complejidad de esta primitiva se encuentra en las llamadas a edgesInSets.

El peor caso en términos de la cantidad de componentes es el mejor caso en cuanto a tamaño de las mismas y viceversa. El análisis de cada caso resulta en que el peor caso tiene un tiempo de ejecución de O(V2)

#comp \* {#comp \* (tamaño \* tamaño) + (tamaño \* (V-tamaño))}

-El peor caso en términos de cantidad, es cuando se tiene la mayor cantidad de ciclos, cada ciclo tiene un mínimo de 3 vértices por lo que, como máximo, habrá V/3 ciclos de tamaño 3

V/3 \* {V/3 \* (3 \* 3) + (3 \* V-3)}

Aplicando propiedades de O(), el tiempo de ejecución es O(V2)

-El peor caso en términos del tamaño es cuando se tiene sólo 2 ciclos, cada uno con la mitad de los vértices.

2 \* {2 (V/2 \* V/2) + (V/2 \* V/2)}

Aplicando propiedades de O(), el tiempo de ejecución es O(V2)

# **Compilación y ejecución**

Al ser Python un lenguaje interpretado, no requiere compilación, directamente se ejecuta el programa mediante la llamada al intérprete con los parámetros necesarios.

La línea de ejecución se indica a continuación:

python tdatp1 <n><archivo>

Siendo <n>, la robustez deseada para el grafo final y <archivo>, el archivo de texto que contiene la descripción del grafo, según el formato descripto en el enunciado.

Para poder ejecutar el programa, sólo es necesario contar con el intérprete de Python instalado.  
  
Siendo que se plantea el requisito de poder ser ejecutado en Linux, como adicional se incluyó un script en bash con un conjunto de casos de prueba ya cargados.

# **Conclusiones**

Se diseño e implementó un algoritmo para fortalecer la conectividad de un grafo, mediante la modificación del recorrido en profundidad, conocido como DFS para identificar las distintas componentes fuertemente conexas de un grafo y luego construir los puentes necesarios entre ellas para dar la robustez necesaria al mismo.

Al solicitar que cada vértice tenga como grado la robustez indicada, se fuerza la condición de robustez para cada componente fuertemente conexa en particular.

Por otra parte, para unir todas las componentes en un grafo de robustez N, se procede a agregar las aristas necesarias, de manera que para aislar alguna de las componentes, se deba eliminar N aristas, cumpliendo así la condición solicitada.

El algoritmo resultante trabaja, en el peor caso, en un orden O (V2), siendo V el número de vértices, aunque en general, el tiempo utilizado será menor.

Existen formas de optimizar el tiempo real de ejecución, pero siendo el peor caso por cada vértice tener que añadir aristas hacia todos los demás vértices no es posible realizarlo en un tiempo mejor al cuadrático (lineal).

# **Código fuente**

**tda1tp.py**

#!/usr/bin/python

"""Prueba cargar un grafo desde un archivo y lo imprime"""

import sys

from Graph import \*

def usage():

print "Usage: python test.py #strenght <filename>"

print "Usage: ./test.py #strenght <filename>"

def checkConditions(graph,strenght):

"""As the graph should be simple and connected, at the most

each vertex on graph will have V-1 edges (complete graph case)

"""

v=graph.countVertices()

maximo = (v-1)

if(int(strenght)>maximo):

print "No es posible la robustez pedida, el maximo es %s, ver condiciones" % maximo

return False

return True

def loadGraph():

graph = Graph()

try:

aux = open(sys.argv[2])

for line in aux:

line = line.replace(':',',').replace('\n','').split(',')

v1 = Vertex(line[0])

graph.addVertex(v1)

for n in range(1,len(line)):

v2 = Vertex(line[n])

if (not graph.isVertex(v2)):

graph.addVertex(v2)

graph.addEdge(v1,v2)

return graph

except KeyError:

print "An error has occurred"

except IndexError:

usage()

except IOError:

print "Verify parameters, input file does not exist"

return False

def main():

print "Teoria y Algoritmos 1 - [75.29]"

print "TP1 - Robustez en Grafos"

print "Autores: Alejandro Pernin (92216) y Lautaro Medrano (90009)\n"

print "Grafo ingresado:\n"

g = loadGraph()

if not g:

return

print g

robustez = sys.argv[1]

print "Se requiere robustez %s" % robustez

if checkConditions(g,robustez):

#print "probando dfs\n"

g.rdfs()

else:

return

g.setStrenght(int(robustez))

print "Resultado:"

g.printResult()

if \_\_name\_\_ == '\_\_main\_\_':

main()

**graph.py**

"""Graph.py

Description: simple & undirected graph implementation for course 75.29 Teoria de

Algoritmos @ University of Buenos Aires

Authors:

Medrano, Lautaro

Pernin, Alejandro

"""

class Vertex:

def \_\_init\_\_(self, key):

self.key = key

#To be used on DFS

self.father = None

self.visited = False

def \_\_eq\_\_(self,other):

return self.key == other.key

def \_\_str\_\_(self):

return self.key

def isVisited(self):

return self.visited

def visit(self,father):

if not father:

f=''

else:

f= father.key

#print "Visitando %s desde %s" % (self.key, f)

self.visited=True

self.father=father

class Graph:

def \_\_init\_\_(self):

#Standalone initialization

self.vertices = {}

self.adjacencies = {}

self.comp = []

self.res = []

def \_\_str\_\_(self):

#Prints graph as 'vertex: [edges]'

s = ''

for v in self.vertices.keys():

#First character on the line is the vertex

s += "%s: " % v

for a in self.adjacencies[v]:

s += "%s," % a

s += "\n"

return s

def printResult(self):

count = 1

for line in self.res:

print "Arista%s: %s" % (count,line)

count += 1

if count == 1:

#There were no results

print "Ya se cumple la robustez requerida"

def countVertices(self):

return len(self.getAllVertex())

def isVertex(self, vertex):

return self.vertices.has\_key(vertex.key)

def addVertex(self, vertex):

if not self.isVertex(vertex):

self.vertices[vertex.key] = vertex

self.adjacencies[vertex.key] = {}

return True

else:

return False

def delVertex(self, vertex):

if self.isVertex(vertex):

self.vertices.pop(vertex.key)

self.adjacencies.pop(vertex.key)

#Now I must delete all adjacencies references to the vertex

for lin in self.adjacencies.values():

try:

l.remove(vertex)

except ValueError:

#DoNothing

continue

def getAllVertex(self):

#Returns all the vertices

return self.vertices.values()

def getAllNeighbours(self, vertex):

#Returns list containing all the neighbours keys of the vertex

if self.isVertex(vertex):

return self.adjacencies[vertex.key].keys()

else:

print "vertex not in grahp"

return False

def addAdjacency(self, vertex, adj):

#Checks if it isnt already a adjacency.

a = self.vertices[adj.key] #O(1)

if not self.adjacencies[vertex.key].has\_key(a.key):

self.adjacencies[vertex.key][a.key]=None

return True

else:

return False

def isEdge(self,vertex1,vertex2):

#print "Chequeando adyacencia entre %s y %s" % (vertex1,vertex2)

if self.adjacencies[vertex2.key].has\_key(vertex1.key):

return True

else:

return False

def addEdge(self, vertex1, vertex2):

#First checks if there is vertex exists

if (self.isVertex(vertex1) and self.isVertex(vertex2)):

#As the intended graph is undirected, A>B and B>A should be made.

self.addAdjacency(vertex1,vertex2)

self.addAdjacency(vertex2,vertex1)

return True

else:

return False

def delEdge(self, vertex1, vertex2):

#As it is undirected, both references must be deleted

try:

self.graph[vertex2.value].pop(vertex1.value)

self.graph[vertex1.value].pop(vertex2.value)

return True

except KeyError:

print "Error while deleting edge"

return False

def rdfs(self):

l=[]

for vert in self.getAllVertex():

if not vert.isVisited():

self.dfsvisit(None,vert,l)

def dfsvisit(self,father,vert,l):

vert.visit(father)

l.append(vert)

for key\_vecino in self.getAllNeighbours(vert):

vecino = self.vertices[key\_vecino]

if not vecino.isVisited():

self.dfsvisit(vert,vecino,l)

elif (vert.father != None) and (vert.father != vecino):

"""If the vertex's neighbour im trying to visit is already visited

and it is not the vertex's father, It means there I have reached a loop.

So I proceed to save the loop """

aux = []

"""As the only time a element is included in the list is when visited,

at most the list will include all the vertices"""

while len(l)>0:

v = l.pop()

aux.append(v)

if v.key == vecino.key:

break

if aux:

self.comp.append(aux)

def addEdgesSets(self,laux,cant):

for i in range(cant):

try:

v1 = laux[i][0]

v2 = laux[i][1]

self.addEdge(v1,v2)

s = "%s , %s" % (v1.key , v2.key)

self.res.append(s)

except IndexError:

#There are no more available edges to add

return

def addEdgesVertex(self,vertex,cant):

for i in self.getAllVertex():

if not cant:

#No edges to add

return

if i == vertex :

continue

elif not self.isEdge(vertex,i):

self.addEdge(vertex,i)

s = "%s , %s" % (vertex.key , i.key)

self.res.append(s)

cant -= 1

def edgesInSets(self,set1,set2,laux):

count=0

for i in set1:

for j in set2:

if self.isEdge(i,j):

count +=1

else:

#Saves the no-edges to be used if more edges are required

laux.append((i,j))

return count

def setStrenght(self,svalue):

cant = len(self.comp)

res = 0

aux = {}

#First analyze the strength from the set point of view (see report)

for li in self.comp:

cont = 0

laux = []

for lj in self.comp:

#See how many edges to other sets there are

if li==lj:

continue

else:

cont += self.edgesInSets(li,lj,laux

if cont < svalue:

#The set has less edges to other sets than the required strength

cant = svalue - cont #The required ammount of edges to be added to comply the requirements

self.addEdgesSets(laux,cant)

#Now analize the strenght from the vertex point of view:

for vertex in self.getAllVertex():

ncount = len(self.getAllNeighbours(vertex))

if (ncount < svalue):

#The vertex has less neighbours than the required strength

cant = svalue - ncount #Ammount of edges to be added

self.addEdgesVertex(vertex,cant)

1. Sea el grafo un grafo no-dirigido, por definición cada componente conexa es una componente fuertemente conexa. En el algoritmo asumimos una componente fuertemente conexa es un ciclo. [↑](#footnote-ref-2)