Universidad de Buenos Aires

Facultad De Ingeniería

Año 2017 - 1er Cuatrimestre

Teoría de Algoritmos I

Trabajo Práctico 2

TEMA: Sociedades

FECHA: 24 de abril de 2017

INTEGRANTES:

KEKLIKIAN, Nicolas - #96480

<nkeklikian@gmail.com>

GUZZARDI, Gonzalo - #94258

< gonzaloguzzardi@gmail.com>

COVA, Alejo - #94325

<covaalejo@gmail.com>

Índice

1. Asignación de residencias

1.1. Reducción al problema de los matrimonios

Se consideran a los estudiantes como hombres, a los hospitales como mujeres. Por cada vacante después de la primera en los hospitales se agrega una copia de dicha mujer, con el mismo orden de merito, y se agrega esta copia a las listas de los hombres, en la posición siguiente a la original. Al obtener el resultado, basta con reemplazar las copias por sus originales para tener la solución al problema original.

1.2. Conclusiones

Se puede observar que la relación del tiempo de procesamiento con el tamaño de la muestra se puede acotar por una función de orden cuadrático.

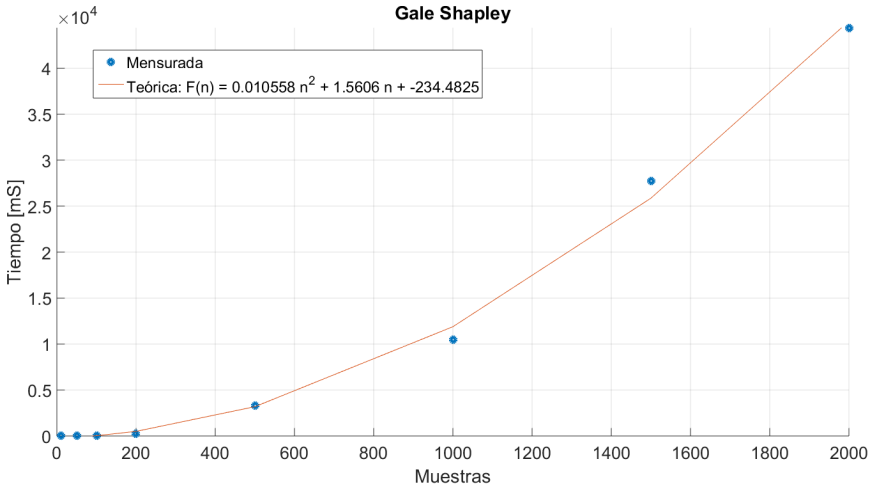

Figura 1: Comparación de muestras con una función proporcional al orden $O(n^2)$.

Es importante mencionar, que el algoritmo planteado resuelve el problema para muestras de hasta, aproximadamente, 2000 aplicantes (n) y 2000 hospitales (m). A valores superiores, la solución propuesta no puede inicializar la matriz de orden de mérito (H), ocasionando la culminación prematura del programa. Una matriz de 10000 x 10000 variables enteras de 32 bits, ocupa 400 MB y debería poder ser resuelta, por lo que se cree que el problema encontrado reside en limitaciones del entorno de desarrollo, estableciendo límites al tamaño del Heap que puede utilizar un programa.

1.3. Instrucciones de ejecución

```
$ python asignacion_de_residencias.py
```

2. Puntos de falla

2.1. Algoritmo

Para encontrar los puntos de articulación de un grafo y lograr un algoritmo de complejidad O(V+E) se utiliza el algoritmo DFS (Depth First Search). Con el recorrido de un grafo utilizando DFS, uno calcula el tiempo de aparición de un vértice y el tiempo de baja (instante en el que el vértice avanza por una arista de retroceso) y con dichos datos se considera que el vértice μ es un punto de articulación si y solo si:

1. Si μ es raíz del árbol DFS y tiene al menos dos hijos.
2. μ no es raíz del árbol y tiene un hijo ν, en donde el subarbol con ν como raíz no tiene una arista de retroceso hacia algún ancestro en el árbol DFS de μ.

Se puede ver que el algoritmo recorre el grafo solo una vez utilizando un DFS modificado, teniendo un orden O(V) y luego por cada arista existente compara los números de baja y aparición para determinar si es una arista de retroceso, teniendo un orden O(E). Por lo tanto el orden de complejidad es O(V+E), siendo V la cantidad de vértices y E la cantidad de aristas del grafo.

2.2. Conclusiones

A partir del siguiente gráfico, se puede concluir que el algoritmo cumple el análisis teórico de que una función proporcional a O(E+V) acota superiormente el comportamiento del tiempo de ejecución para distintas dimensiones del grafo.

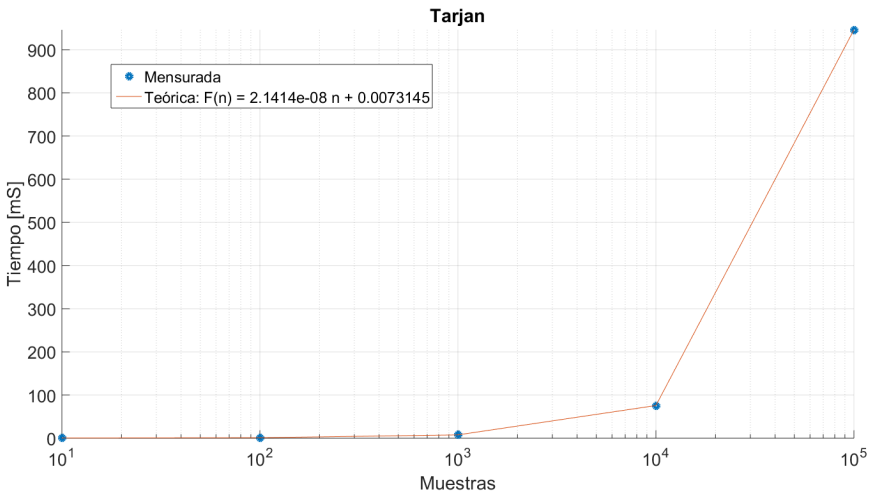

Figura 2: Comparación de muestras con una función proporcional al orden O(E+V).

Es menester aclarar, que no se pudo obtener una solución utilizando como input el archivo g6.txt. Esto se debe a que para el tamaño del input propuesto, el programa excede los límites de llamadas recursivas permitidas en nuestro entorno de desarrollo y, por ende, el programa finaliza prematuramente sin obtener solución. Para los demás archivos, el programa se ejecutó y encontró solución correctamente.

2.3. Instrucciones de ejecución

```
$ python puntos_de_falla.py
```

3. Comunidades en redes

3.1. Algoritmo

Sea $G = (V, E)$ un grafo dirigido:

1. Aplicar búsqueda en profundidad sobre G.
2. Calcular el grafo traspuesto G^t. Aplicar búsqueda en profundidad sobre G^t (el grafo traspuesto) iniciando la búsqueda en los nodos de mayor a menor tiempo de finalización obtenidos en la primera ejecución de búsqueda en profundidad (paso 1).
3. El resultado será un bosque de árboles. Cada árbol es un componente fuertemente conexo.

Las dos búsquedas en profundidad y la construcción del grafo reverso consumen tiempo lineal, de manera que el tiempo total es también lineal.

3.2. Conclusiones

A partir del siguiente gráfico, se puede concluir que el algoritmo cumple el análisis teórico de que una función proporcional a O(E+V) acota superiormente el comportamiento del tiempo de ejecución para distintas dimensiones del grafo.

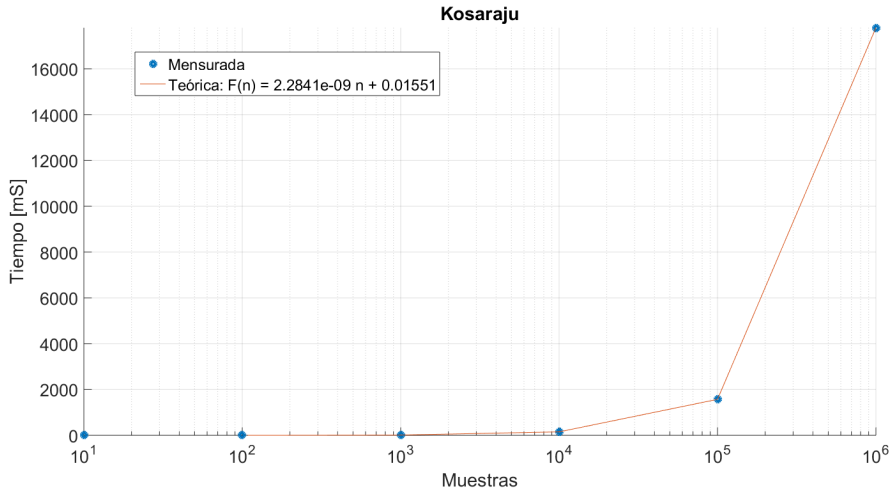

Figura 3: Comparación de muestras con una función proporcional al orden O(E+V).

3.3. Instrucciones de ejecución

```
$ python comunidades_en_redes.py
```

4. Código fuente

4.1. Asignación de residencias

```
from random import shuffle, choice
from collections import defaultdict
from timeit import default_timer as timer

class Residencias:

    def __init__(self, pedir_input=False, cant_estudiantes=100, cant_hospitales=1
        self.resultado = []
        if pedir_input:
            self.estudiantes = input("Ingresar la cantidad de estudiantes: ")
            self.hospitales = input("Ingresar la cantidad de hospitales: ")
            self.vacantes = input("Ingresar la cantidad de vacantes maxima: ")
        else :
            self.estudiantes = cant_estudiantes
            self.hospitales = cant_hospitales
            self.vacantes = cant_vacantes
        self.E = [self.mezclar_lista(range(self.hospitales)) for _ in xrange(self
        self.H = [self.mezclar_lista(range(self.estudiantes)) for _ in xrange(self
        self.Q = [choice(range(1,self.vacantes+1)) for _ in xrange(self.hospitales
        self.equivalencias = {}
        self.escribir_archivo("output_asignacion_de_residencias.txt")

    def mezclar_lista(self, l): ## Agarra una lista y la devuelve desordenada
        shuffle(l)
        return l

    def preferencias_estudiantes(self):
        return self.E
```

```
def orden_merito(self):
    return self.H

def vacantes(self):
    return self.Q

def reducir(self):
    from copy import deepcopy
    hombres = defaultdict(list)
    mujeres = deepcopy(self.H)
    dic_m = defaultdict(list)
    m_cant = self.hospitales
    for m in xrange(self.hospitales):
        dic_m[m].append(m)
        for _ in xrange(1, self.Q[m]):
            dic_m[m].append(m_cant)
            mujeres.append(mujeres[m])
            self.equivalencias[m_cant] = m
            m_cant += 1
    for h in xrange(self.estudiantes):
        for m in xrange(self.hospitales):
            hombres[h].extend(dic_m[self.E[h][m]])
    return (self.estudiantes, m_cant, [hombres[h] for h in xrange(self.estudia

def gale_shapley(self, hombres, mujeres, pref_hombre, pref_mujer):
    prop_mujer = defaultdict(list)
    pareja_hombre = [-1] * hombres
    sig_prop_hombre = [0] * hombres
    while -1 in pareja_hombre:
        for h in xrange(hombres):
            if pareja_hombre[h] == -1:
                prop_mujer[pref_hombre[h][sig_prop_hombre[h]]].append(h)
```

```
            for m in xrange(mujeres):
                props_m = prop_mujer[m]
                if props_m:
                    props_m.sort(key = lambda x : pref_mujer[m].index(x))
                    pareja_hombre[props_m[0]] = m
                    for i in xrange(1, len(props_m)):
                        pareja_hombre[props_m[i]] = -1
                        sig_prop_hombre[props_m[i]] += 1
                    prop_mujer[m] = [props_m[0]]
        return [pref_hombre[i][sig_prop_hombre[i]] for i in xrange(hombres)]

    def obtener_resultados(self):
        self.resultado = self.gale_shapley(*self.reducir())
        for i in xrange(len(self.resultado)):
            if self.resultado[i] in self.equivalencias:
                self.resultado[i] = self.equivalencias[self.resultado[i]]
        return self.resultado

    def escribir_archivo(self, ruta):
        with open("output_asignacion_de_residencias.txt","w") as archivo:
            archivo.write(str(self.estudiantes) + "\n")
            for estudiante in self.E:
                archivo.write("".join(str(pref) + "␣" for pref in estudiante) + "\
            archivo.write(str(self.hospitales) + "\n")
            for hospital in self.H:
                archivo.write("".join(str(merito) + "␣" for merito in hospital) +
            archivo.write("".join(str(vac) + "␣" for vac in self.Q) + "\n")

start = timer()

r1 = Residencias(False,1000,1000,1)
print r1.obtener_resultados()
```

```
end = timer()

print("Hospitales: "+str(r1.hospitales))
print("Estudiantes: "+str(r1.estudiantes))
print("Tiempo: "+str((end - start)*1000)+" mseg")
```

4.2. Puntos de falla

```
from GrafoUtils import Grafo, parse

class Tarjan:
    def __init__(self, grafo):
        visitado = [False] * (grafo.vertices())
        tiempo_de_descubrimiento = [float("Inf")] * (grafo.vertices())
        tiempo_de_baja = [float("Inf")] * (grafo.vertices())
        padre = [-1] * (grafo.vertices())
        punto_articulacion = [False] * (grafo.vertices())
        self.tiempo = 0                    #Tiempo de encuentro

        def visitar(u):

            hijos = 0
            visitado[u] = True
            tiempo_de_descubrimiento[u] = self.tiempo
            tiempo_de_baja[u] = self.tiempo
            self.tiempo += 1

            for v in grafo.vecinos(u):
                if not visitado[v] :
                    padre[v] = u
                    hijos += 1
                    visitar(v) #Llamada recursiva de la funcion
```

```
                    tiempo_de_baja[u] = min(tiempo_de_baja[u], tiempo_de_baja[v])

                    if padre[u] == -1 and hijos > 1:
                        punto_articulacion[u] = True

                    if padre[u] != -1 and tiempo_de_baja[v] >= tiempo_de_descubri
                        punto_articulacion[u] = True

                elif v != padre[u]:
                    tiempo_de_baja[u] = min(tiempo_de_baja[u], tiempo_de_descubri

        for v in xrange(grafo.vertices()):

            if not visitado[v]:
                visitar(v)

        self.puntos_articulacion = [v for v in xrange(grafo.vertices()) if punto_

    def get_puntos_articulacion(self):
        return self.puntos_articulacion

# Ejemplo
from timeit import default_timer as timer

start = timer()

from sys import setrecursionlimit

setrecursionlimit(10000)

t1 = Tarjan(parse(Grafo, "Archivos/Problema␣2/g6.txt"))
print "Puntos␣de␣articulacion:␣" + "".join(str(v) + "␣" for v in t1.get_puntos_ar
```

```
end = timer()

print(str((end - start)*1000)+" mseg")
```

4.3. Comunidades en Redes

```
from collections import deque ## cola optimizada
from collections import defaultdict
from GrafoUtils import Digrafo, parse

class Kosaraju:

    def __init__(self, grafo):
        self.grafo = grafo
        self.componentes = defaultdict(list)
        self.componentes_fuertemente_conexas(self.grafo.vertices())

    def componentes_fuertemente_conexas(self, vertices):
        ## Devuelve la cantidad de componentes conexas
        ## Utilizando el algoritmo de Kosaraju
        visitado = [False] * vertices
        L = deque([])

        def visitar(u):
            if not visitado[u]:
                visitado[u] = True
                for v in self.grafo.vecinos_entrantes(u):
                    visitar(v)
                L.appendleft(u)

        for u in xrange(vertices):
            visitar(u)
```

```
        asignado = [-1] * vertices

        def asignar(u, raiz):
            if asignado[u] == -1:
                asignado[u] = raiz
                for v in self.grafo.vecinos(u):
                    asignar(v, raiz)
        for u in L:
            asignar(u,u)

        for i in xrange(vertices):
            self.componentes[asignado[i]].append(i)

    def cantidad_componentes(self):
        return len(self.componentes)

    def componente_dado_elemento(self, elemento):
        return self.componentes[self.asignado[elemento]]

# Ejemplo
from timeit import default_timer as timer

start = timer()

k1 = Kosaraju(parse(Digrafo, "Archivos/Problema 3/d6.txt"))
print k1.cantidad_componentes()

end = timer()

print(str((end - start)*1000)+" mseg")
```