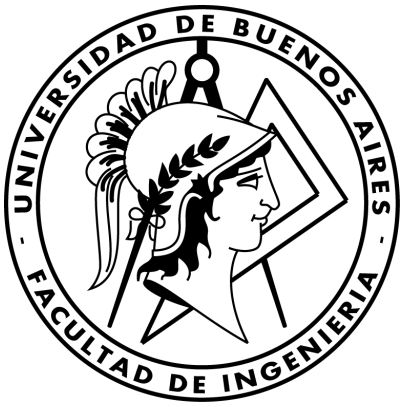

Trabajo Práctico N°1

75.29 - Teoría de Algoritmos I

Facultad de Ingeniería de la Universidad de Buenos Aires

1er. Cuatrimestre 2017

Federico Brasburg, *Padrón Nro. 96.653*
federico.brasburg.@gmail.com

Pablo Rodrigo Ciruzzi, *Padrón Nro. 95.748*
p.ciruzzi@hotmail.com

Andrés Otero, *Padrón Nro. 96.604*
oteroandres95@gmail.com

24 de abril de 2017

Índice

1. Asignación de residencias

1.1. Reducción

Para reducir el problema de asignación de residencias al de matrimonios estables decidimos tomar algunas hipótesis para simplificar el problema y hacer más fácil la reducción. Las hipótesis son que el número de hospitales debe ser menor o igual al de residentes, así todos los hospitales tienen una vacante. El problema es que distintos hospitales tienen diferente cantidad de vacantes entonces para reducirlo hicimos que se modele cada vacante como un hospital diferente que tiene el mismo "ranking" que el hospital de la vacante. De esa manera se vuelve un problema de n estudiantes contra $\sum_{i=0}^{|Q|-1} Q[i]$ hospitales (o vacantes).

1.2. Resultados

El problema de asignación n = m = 100 tardó 0.00866603851318 segundos
El problema de asignación n = m = 1000 tardó 0.756355047226 segundos
El problema de asignación n = m = 10000 tardó 85.6384279728 segundos
El problema de asignación n = m = 100000 no se corrió

1.3. Conclusiones

El primer comentario a hacer es la no realización de n = m = 100000; se empezó corriendo pero se notaron problemas para hacerlo. Luego recurrimos a hacer las cuentas para dimensionar el problema y, al ser uno de 100000x100000, se necesitarían 10.000 millones de enteros para dimensionar una de las matrices de rankings. Es por ello que se decidió no realizar esta iteración del problema.

En cuanto al rendimiento respecto al crecimiento del problema, vale la pena observar que el orden del problema $O(n^2)$ coincide con el crecimiento de los tiempos respecto del orden de n, es decir, subiendo un orden el n se puede apreciar que sube 2 ordenes el tiempo utilizado. Podemos arriesgar que el problema más grande que no pudimos realizar estaría en el orden de 2 horas y media, suponiendo que el rendimiento fuera igual al de los anteriores.

1.4. Cómo correrlo

Para crear y resolver un problema, se puede simplemente usar el script `correr_asignacion.py` en la carpeta `src`. Para saber como usarlo correr `python correr_asignacion.py -h`.

2. Puntos de falla

2.1. Resultados

Con 10 vértices y 20 aristas, tardó: 0.000194072723389 segundos
Con 100 vértices y 200 aristas, tardó: 0.00134420394897 segundos
Con 1000 vértices y 2000 aristas, tardó: 0.015242099762 segundos
Con 10000 vértices y 20000 aristas, tardó: 0.146757125854 segundos
Con 100000 vértices y 200000 aristas, tardó: 2.14698195457 segundos
Con 1000000 vértices y 2000000 aristas, tardó: 25.5380079746 segundos

Cabe aclarar que la cantidad de aristas está multiplicada por 2 ya que en nuestro modelo una arista no dirigida se corresponde a 2 aristas dirigidas.

Aquí no aparece como resultado, pero se agregó un grafo de 7 vértices analizado en el curso. Se agregó de dos maneras distintas (dando dos numeraciones distintas a los vértices) para comprobar el buen funcionamiento del algoritmo ante el caso de una raíz con más de un hijo. Estos se encuentra en los archivos `g0.txt` y `g7.txt`.

2.2. Funcionamiento

Cabe mencionar que en un primer momento el algoritmo se hizo de forma recursiva. El problema surgió cuando de esta manera, para los problemas más grandes, se excedía la cantidad de llamadas recursivas posibles dentro de Python. Por ello se tuvo que pasar a una forma iterativa del mismo.

El funcionamiento en sí del algoritmo es bastante simple y se basa en recorridos DFS de un grafo. Lo que se hace es hacer un primer DFS para asignar el orden de visita de los nodos. Luego, utilizando este recorrido (modelado en una pila en nuestro caso), se asignan los valores del *bajo* para cada nodo. En este segundo recorrido, haciendo algunas comparaciones con este valor, se obtienen los vértices que son puntos de articulación.

Por último, al terminar de recorrer un árbol DFS, se contempla el caso especial de la raíz, el cual se debe ver si tiene 2 o más hijos dentro del árbol para ver si es realmente punto de articulación o no.

2.3. Conclusiones

Lo más importante a destacar es que se puede notar el orden lineal del algoritmo ($O(|V|+|E|)$) mirando los tiempos de ejecución. Es simple ver cómo, cuando se aumenta en 1 el orden de magnitud de la entrada, lo mismo sucede con el tiempo de ejecución.

2.4. Cómo correrlo

Para resolver los problemas brindados por el curso, basta con correr `python puntos_articulacion.py`.

3. Comunidades en redes

3.1. Algoritmo

Para esta parte del práctico, se implementó el algoritmo de Kosaraju que, dado un grafo G dirigido, calcula las componentes fuertemente conexas de G de la siguiente manera:

- Realiza un DFS a G y almacena el tiempo de finalizado de los vértices.
- Traspone G
- Realiza un DFS a G traspuesta respetando el orden de forma descendiente del tiempo de finalizado almacenado cuando se requiera decidir por cual vértice seguir.

3.2. Resultados

Con 10 vértices y 20 aristas, tardó: 0.000113964080811 segundos
Con 100 vértices y 250 aristas, tardó: 0.00108599662781 segundos
Con 1000 vértices y 2500 aristas, tardó: 0.0141451358795 segundos
Con 10000 vértices y 25000 aristas, tardó: 0.148002147675 segundos
Con 100000 vértices y 250000 aristas, tardó: 2.38693785667 segundos
Con 1000000 vértices y 2500000 aristas, tardó: 31.3342020512 segundos

3.3. Conclusiones

Se puede ver en los resultados que el orden del algoritmo de Kosaraju (Que es $O(|V|+|E|)$), al estar implementado con un diccionario, es lineal y el orden crecimiento en tiempo del algoritmo implementado también lo es. Finalmente, se comprobó que el orden lineal del algoritmo es correcto.

3.4. Cómo correrlo

Correr la resolución de los problemas de componentes fuertemente conexas es tan simple como correr `python CFC.py`.

4. Código

asignacion_de_residencias.py

```
from random import shuffle
from random import sample
from parser import Parser
from collections import deque


def crear_archivo_problema(nombre, m, n):
    parser = Parser()
    E, H, Q = crear_problema(int(m), int(n))
    parser.escribir_stable_matching(nombre, E, H, Q)

def crear_problema(m, n):
    if m > n:
        print "El numero de hospitales debe ser menor o igual al de pacientes"
        exit()
    E = crear_lista_de_listas_al_azar(m, n)
    H = crear_lista_de_listas_al_azar(n, m)
    Q = constrained_sum_sample_pos(m, n)
    return E, H, Q

def crear_lista_de_listas_al_azar(cantidad_de_elementos, cantidad_de_listas):
    lista = [[i for i in range(cantidad_de_elementos)] for l in range(cantidad_de_listas)]
    [shuffle(l) for l in lista]
    return lista

def resolver_archivo_problema(archivo):
    parser = Parser()
    E, H, Q = parser.leer_stable_matching(archivo)
    return resolver_problema(E, H, Q)

def resolver_archivo_problema_con_archivo_salida(archivo_problema, archivo_salida):
    parser = Parser()
    E, H, Q = parser.leer_stable_matching(archivo_problema)
    P = resolver_problema(E, H, Q)
    f = open(archivo_salida, 'w')
    f.write(str(P))
    f.close()
    return True

def constrained_sum_sample_pos(n, total):
    # http://stackoverflow.com/questions/3589214/generate-multiple-random-numbers-to-equal-a
        -value-in-python
    """Return a randomly chosen list of n positive integers summing to total.
    Each such list is equally likely to occur."""
    dividers = sorted(sample(xrange(1, total), n - 1))
    return [a - b for a, b in zip(dividers + [total], [0] + dividers)]

def reducir_problema(E, H, Q):
    viejo_E = E
    nuevo_E = []
    nuevo_H = H
    for i in range(0, len(Q)):
```

```
        nuevo_E = []
        tam_H = len(nuevo_H)
        nuevo_H = nuevo_H + [nuevo_H[i]] * (Q[i] - 1)
        nuevo_tam_H = len(nuevo_H)
        for l in viejo_E:
            nuevo_E += [l[:l.index(i) + 1] + range(tam_H, nuevo_tam_H) + l[l.index(i) + 1:]]
        viejo_E = nuevo_E
    return nuevo_E, nuevo_H

def resolver_problema(E, H, Q):
    n = len(E)
    m = len(H)
    if m > n:
        print "El numero de hospitales debe ser menor o igual al de pacientes"
        exit()
    if n != m:
        E, H = reducir_problema(E, H, Q)

    sig_deseado = [0] * n
    P = [None] * n
    pendientes = deque(range(n))
    while len(pendientes) != 0:
        e = pendientes.pop()
        h_deseado = E[e][sig_deseado[e]]
        sig_deseado[e] += 1

        e_rival = P[h_deseado]
        if e_rival is None:
            P[h_deseado] = e
        elif H[h_deseado][e] > H[h_deseado][e_rival]:
            P[h_deseado] = e
            pendientes.append(e_rival)
        else:
            pendientes.append(e)
    return P
```

CFC.py

```
from parser import Parser
from grafo import trasponer
from dfs import DFS
import time

"""Utilizar como python CFC.py"""

def CFC(g):
    """Recibe un grafo y devuelve las componentes fuertemente conexas"""
    r = DFS(g)
    g_traspuesta = trasponer(g)
    orden = sorted(r.get_tiempo_finalizado(), key=r.get_tiempo_finalizado().get, reverse=True)
    return DFS(g_traspuesta, orden).get_bosque_DFS()

for i in [0, 1, 2, 3, 4, 5, 6]:
    parser = Parser()
    g = parser.leer_grafo_dirigido("../in/ej3/d"+str(i)+".txt")
    start = time.time()
```

```
    CFC(g)
    end = time.time()
    print("Con " + str(g.devolver_cant_vertices()) + " vertices y " + str(len(g.devolver_aristas()
        )) + " aristas, tardo: " + str(end - start) + " segundos")
```

correr_asignacion.py

```
import sys
import os
import time
from asignacion_de_residencias import *

if len(sys.argv) != 5:
    if len(sys.argv) == 2 and (sys.argv[1] == "-h" or sys.argv[1] == "--help"):
        print "\nPara correr el programa se necesitan 4 parametros:\n" \
              "\t-1.El nombre del archivo a crearse en la entrada\n" \
              "\t-2.El numero m de hospitales\n" \
              "\t-3.El numero n de pacientes\n" \
              "\t-4.El nombre del archivo de salida\n" \
              "Un ejemplo que corre correctamente es (desde la carpeta src):\n" \
              "\tpython correr_asignacion.py 'in.txt' 100 100 'out.txt'\n " \
              "Recuerde revisar en las carpetas in/ y out/ para ver los archivos correspondientes"
    else:
        print "Revise que esten bien las entradas. Para mas ayuda ingrese -h o --help"
    exit()
cwd = os.getcwd().split('/')
if cwd[len(cwd) - 1] != "src":
    print "El programa debe ser corrido desde la carpeta 'src' del proyecto"
    exit()

nombre_archivo_problema = "../in/" + sys.argv[1]
nombre_archivo_salida = "../out/" + sys.argv[4]
m = sys.argv[2]
n = sys.argv[3]
crear_archivo_problema(nombre_archivo_problema, m, n)
start = time.time()
resolver_archivo_problema_con_archivo_salida(nombre_archivo_problema, nombre_archivo_salida
    )
end = time.time()
print("El problema de asignacion " + sys.argv[1] + " tardo " + str(end - start))
```

dfs.py

```
from tiempo import Tiempo
from resultado_DFS import ResultadoDFS


def DFS(g, lista_vertices={}):
        if lista_vertices == {}:
            lista_vertices = g.devolver_vertices()
        visitado = {}
        tiempo_visitado = {}
        tiempo = Tiempo()
        bosque = []
        f = {}
        for v in lista_vertices:
```

```
            visitado[v] = False
        for v in lista_vertices:
            if not visitado[v]:
                arbol = []
                DFS_Visitar(g, v, visitado, tiempo, tiempo_visitado, f, arbol)
                bosque.append(arbol)
        return ResultadoDFS(tiempo_visitado, f, bosque)

def DFS_Visitar(g, v, visitado, tiempo, tiempo_visitado, f, arbol):
        visitado[v] = True
        arbol.append(v)
        tiempo.incrementar()
        tiempo_visitado[v] = tiempo.actual()
        for u in g.adyacentes(v):
            if not visitado[u]:
                DFS_Visitar(g, u, visitado, tiempo, tiempo_visitado, f, arbol)
        tiempo.incrementar()
        f[v] = tiempo.actual()
```

dfs_iterativo.py

```
from tiempo import Tiempo


class DFSIterativo(object):

    def __init__(self, g, v, visitado):
        self.g = g
        self.v = v
        self.visitado = visitado
        self.tiempo_visitado = {}
        self.tiempo = Tiempo()
        self.predecesor = {}
        self.bajo = {}
        self.puntos_articulacion = set()
        for u in g.devolver_vertices():
            self.predecesor[u] = None

    def _asignar_visitado(self):
        stack = [self.v]
        stack_recorrido = [self.v]
        while stack:
            u = stack.pop()
            if self.visitado[u]:
                continue
            self.visitado[u] = True
            self.tiempo.incrementar()
            self.tiempo_visitado[u] = self.tiempo.actual()
            for w in self.g.adyacentes(u):
                if not self.visitado[w]:
                    stack.append(w)
                    stack_recorrido.append(w)
                    self.predecesor[w] = u
        return stack_recorrido

    def _asignar_bajo(self, stack):
```

```
        while stack:
            u = stack.pop()
            self.bajo[u] = self.tiempo_visitado[u]
            for w in self.g.adyacentes(u):
                if self.tiempo_visitado[w] > self.tiempo_visitado[u]:
                    if self.bajo[w] >= self.tiempo_visitado[u]:
                        self.puntos_articulacion.add(u)
                    self.bajo[u] = min(self.bajo[u], self.bajo[w])
                elif w != self.predecesor[u]:
                    self.bajo[u] = min(self.bajo[u], self.tiempo_visitado[w])

    def get_predecesor(self):
        return self.predecesor

    def get_puntos_articulacion(self):
        return self.puntos_articulacion

    def hacer_dfs(self):
        stack_recorrido = self._asignar_visitado()
        self._asignar_bajo(stack_recorrido)
```

grafo.py

```
PRIMERO = 0
SEGUNDO = 1


class Arista(object):
    def __init__(self, id1, id2, peso):
        self.id1 = id1
        self.id2 = id2
        self.peso = peso

    def peso(self):
        return self.peso

    def __str__(self):
        return str(self.id1) + " a " + str(self.id2) + ", peso " + str(self.peso)


def trasponer(g):
        """Traspone el mismo grafo"""
        g_t = Grafo()
        for vertice in g.devolver_vertices():
            g_t.agregar_vertice(vertice)
        for arista in g.devolver_aristas():
            g_t.agregar_arista_dirigida(arista.id2, arista.id1, arista.peso)
        return g_t


class Grafo(object):
    def __init__(self):
        """Crea un Grafo dirigido (o no) con aristas pesadas (o no)"""
        self.aristas = {}
        self.vertices = []
```

```
def devolver_aristas(self):
    """Devuelve las aristas del grafo"""
    lista_aristas = []
    for dic_aristas in self.aristas.values():
        for aristas in dic_aristas.values():
            lista_aristas += [aristas]
    return lista_aristas

def devolver_vertices(self):
    return self.vertices

def devolver_cant_vertices(self):
    """Devuelve los nodos del grafo"""
    return len(self.vertices)

def agregar_vertice(self, id):
    """Agrega un vertice que se identifica con un nombre y un ID"""
    self.vertices.append(id)
    self.aristas[id] = {}

def agregar_arista_no_dirigida(self, id1, id2, peso=0):
    """Agrego una arista no dirigida entre los nodos con id1 y id2"""
    self.agregar_arista_dirigida(id1, id2, peso)
    self.agregar_arista_dirigida(id2, id1, peso)

def agregar_arista_dirigida(self, id1, id2, peso=0):
    """Agrego una arista dirigida entre los nodos con id1 y id2"""
    self.aristas[id1][id2] = Arista(id1, id2, peso)

def son_vecinos(self, id1, id2):
    """Devuelve si id1 y id2 son vecinos"""
    return id2 in self.aristas[id1].keys()  # Azucar sintactico

def peso_arista(self, id1, id2):
    """Devuelve el peso de la arista entre id1 e id2"""
    if self.son_vecinos(id1, id2):
        return self.aristas[id1][id2].peso
    raise ValueError

def adyacentes(self, id):
    """Pide un id de un nodo existe y devuelve una lista de los id de sus adyacentes"""
    adyacentes = []
    for arista in self.aristas[id]:
        adyacentes.append(arista)
    return adyacentes

def _leer(self, nombre, dirigido=False):
    """Lee un grafo (dirigido o no) de un archivo con nombre"""
    try:
        mi_arch = open(nombre)
        cant_nodos = int(mi_arch.readline())
        for i in range(0, cant_nodos):
            self.agregar_vertice(i)
        cant_aristas = int(mi_arch.readline())
        for i in range(0, cant_aristas):
            linea = mi_arch.readline()
```

```
                numeros = linea.split("␣")
                numeros[SEGUNDO] = numeros[SEGUNDO].rstrip('\n')
                if dirigido:
                    self.agregar_arista_dirigida(int(numeros[PRIMERO]), int(numeros[SEGUNDO
                        ]))
                else:
                    self.agregar_arista_no_dirigida(int(numeros[PRIMERO]), int(numeros[
                        SEGUNDO]))
            mi_arch.close()
            return True
        except:
            return False

    def leer_dirigido(self, nombre):
        self._leer(nombre, True)

    def leer_no_dirigido(self, nombre):
        self._leer(nombre, False)
```

parser.py

```
from grafo import Grafo

CERO = 0
UNO = 1


class Parser(object):

    def escribir_stable_matching(self, nombre, E, H, Q):
        """Escribe un archivo del tipo Stable Matching"""
        try:
            mi_arch = open(nombre, 'w')
            n = len(E)
            mi_arch.write(str(n) + '\n')
            for i in range(0, n):
                numeros = "␣".join(str(x) for x in E[i]) + '\n'  # Deberia haber m numeros
                mi_arch.write(numeros)
            m = len(H)
            mi_arch.write(str(m) + '\n')
            for i in range(0, m):
                numeros = "␣".join(str(x) for x in H[i]) + '\n'  # Deberia haber n numeros
                mi_arch.write(numeros)
            numeros = "␣".join(str(x) for x in Q) + '\n'  # Deberia haber m numeros
            mi_arch.write(numeros)
            mi_arch.close()
            return True
        except:
            print "Ocurrio␣un␣error␣leyendo␣el␣archivo␣de␣Stable␣Matching␣" + nombre
            return False

    def _read_line(self, mi_arch):
        return mi_arch.readline().strip("\n")

    def _read_line_int_list(self, mi_arch):
        return [int(i) for i in self._read_line(mi_arch).split("␣")]
```

```
    def leer_stable_matching(self, nombre):
        """Escribe un archivo del tipo Stable Matching"""
        try:
            mi_arch = open(nombre, 'r')
            n = int(self._read_line(mi_arch))
            E = []
            for i in range(0, n):
                E.append(self._read_line_int_list(mi_arch))
            m = int(self._read_line(mi_arch))
            H = []
            for i in range(0, m):
                H.append(self._read_line_int_list(mi_arch))
            Q = self._read_line_int_list(mi_arch)
            mi_arch.close()
            return E, H, Q
        except:
            print "Ocurrio un error leyendo el archivo"
            return False

    def leer_grafo_no_dirigido(self, nombre):
        """Lee un archivo de un grafo no dirigido sin peso"""
        try:
            grafo = Grafo()
            grafo.leer_no_dirigido(nombre)
            return grafo
        except:
            print "Ocurrio un error leyendo el archivo de grafo no dirigido " + nombre
            return False

    def leer_grafo_dirigido(self, nombre):
        """Lee un archivo de un grafo dirigido sin peso"""
        try:
            grafo = Grafo()
            grafo.leer_dirigido(nombre)
            return grafo
        except:
            print "Ocurrio un error leyendo el archivo de grafo dirigido " + nombre
            return False
```

puntos_articulacion.py

```
from parser import Parser
from dfs_iterativo import DFSIterativo
import time

"""Utilizar como python puntos_articulacion.py"""


class PuntosArticulacion(object):

    def __init__(self, g):
        self.g = g

    def get_puntos_articulacion(self):
        """Dado un grafo no dirigido, devuelve los puntos de articulacion"""
```

```
        # Inicializo variables
        visitado = {}
        puntos_articulacion = set()
        for u in grafo.devolver_vertices():
            visitado[u] = False

        for v in grafo.devolver_vertices():
            if not visitado[v]:
                # Armo el arbol DFS desde v
                dfs = DFSIterativo(grafo, v, visitado)
                dfs.hacer_dfs()
                puntos_articulacion_v = dfs.get_puntos_articulacion()
                # Como v es raiz del arbol, lo saco para analizarlo por separado
                if v in puntos_articulacion_v: # Aunque deberia estar incluida siempre
                    puntos_articulacion_v.remove(v)
                self.analizar_raiz(dfs.get_predecesor(), puntos_articulacion_v, v)
                puntos_articulacion.update(puntos_articulacion_v)
        return puntos_articulacion

    def analizar_raiz(self, predecesor, puntos_articulacion, v):
        # Analizo la raiz como puntos de articulacion
        hijos = 0
        # Basta con revisar si los adyacentes a v lo tienen como predecesor o no, no es necesario en
            todos los vertices
        for u in self.g.adyacentes(v):
            if predecesor[u] == v:
                hijos += 1
        if hijos >= 2:
            puntos_articulacion.add(v)


for i in [0, 7, 1, 2, 3, 4, 5, 6]:
    start = time.time()
    parser = Parser()
    grafo = parser.leer_grafo_no_dirigido("../in/ej2/g" + str(i) + ".txt")
    puntos_articulacion = PuntosArticulacion(grafo).get_puntos_articulacion()
    print "Hay", len(puntos_articulacion), "puntos de articulacion"
    end = time.time()
    print("Con " + str(grafo.devolver_cant_vertices()) + " vertices y " + str(len(grafo.
        devolver_aristas())) + " aristas, tardo: " + str(end - start) + " segundos")
```

resultado_DFS.py

```
class ResultadoDFS:

    def __init__(self, tiempo_visitado, tiempo_finalizado, bosque):
        self.bosque = bosque
        self.tiempo_finalizado = tiempo_finalizado
        self.tiempo_visitado = tiempo_visitado

    def get_tiempo_visitado(self):
        return self.tiempo_visitado

    def get_tiempo_finalizado(self):
        return self.tiempo_finalizado
```

```
    def get_bosque_DFS(self):
        return self.bosque
```

tiempo.py

```
class Tiempo:

    def __init__(self):
        self.t = 0

    def incrementar(self):
        self.t += 1

    def actual(self):
        return self.t
```