Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Ciencias de la Computación
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS UNIVERSIDAD DE CHILE

Tarea 3

Matemáticas Discretas para la Computación

Autor:	Lukas Pavez
	RUT: 19.401.577-1
Profesor:	Pablo Barcelo B.
Auxiliares:	Arniel Labrada D.
	Ilana Mergudich T.
Ayudantes:	Jaime Salas T.
	Javier Marinković
	Joaquín Romero
	Pablo Paredes Haz

Fecha de entrega: 23/08/2018

1. P1

1.1. Parte a

Sea A la matriz de adyacencia de un grafo dirigido G, que contiene n nodos (enumerados de 1 a n). Se quiere demostrar que A^2 es la matriz que representa los caminos de largo 2 entre 2 nodos.

Se parte por ver la fórmula para calcular cada c_{ij} de A^2:

$$A^2 = A * A \implies c_{ij} = \sum_{r=1}^{n} a_{ir} * a_{rj}$$

Al ver la fórmula se puede notar que para cada nodo r en el grafo, se multiplica $a_{ir} * a_{rj}$, donde la primera parte de la multiplicación ve si el nodo i está conectado con el nodo r, y la segunda parte ve si el nodo r está conectado con el nodo j, esto efectivamente está buscando un camino de largo 2 que contiene las aristas i-r y r-j, si ambas aristas están presentes en el grafo, entonces se suma 1 camino, y como se cuenta para r de 1 a n, se cuenta la totalidad de caminos de largo 2 que hay entre i y j.

Por lo tanto, A^2 es la matriz que representa los caminos de largo 2 entre 2 nodos.

1.2. Parte b

Se quiere demostrar que A^k es la matriz que representa los caminos de largo k entre 2 nodos. Esto se demostrará por inducción:
Caso base k=2: está demostrado en la Parte a. Hipótesis inductiva: A^k es la matriz que representa los caminos de largo k entre 2 nodos.

Paso k+1: Ahora se debe demostrar que A^{k+1} es la matriz que representa los caminos de largo k+1 entre 2 nodos.

Como se sabe que A^k es la matriz que representa los caminos de largo k entre 2 nodos, A^{k+1} se calcula como $A^k * A$ por lo que cada c_{ij} de A^{k+1} quedaría de la siguiente forma:

$$A^{k+1} = A^k * A \implies c_{ij} = \sum_{r=1}^{n} ak_{ir} * a_{rj}$$

Donde ak_{ij} son los elementos de A^k, al ver la fórmula, se puede ver que para cada nodo r del grafo, se ve el número de caminos de largo k que hay entre i y r (se sabe que ak_{ij} es el número de caminos largo k entre i y r por HI), luego se ve si es que hay una arista desde el nodo r y el nodo j, por lo que se tendrían k+1 caminos que pasan por el nodo r, y como se suman los caminos de largo k+1 que pasan para todos los nodos del grafo (r de 1 a n), se llega a que c_{ij} representa todos los caminos de largo k+1 que empiezan en i y terminan en r, lo que implica que A^{k+1} es la matriz que representa los caminos de largo k+1 entre 2 nodos.

Por lo tanto, A^k es la matriz que representa los caminos de largo k entre 2 nodos.

1.3. Código

El lenguaje utilizado es Python 2.7, código adjunto en sección Anexos

El código desarrollado hace lo siguiente:

Al correr el programa, se parte preguntando al usuario por los valores de input, parte preguntando por el número de nodos en el árbol n (enumerados de 1 a n) (por ejemplo n=5), después pregunta por la matriz de adyacencia, donde se deben ingresar las filas separadas por \n, los elementos en cada fila estan separados por un espacio (un ejemplo de input es 0 0 1 0 1\n0 0 1 0 0\n1 1 0 1 0\n0 0 1 0 0\n1 0 0 0 0), luego pregunta por el largo k deseado (ej: k=2), finalmente pregunta por el par de nodos deseados, estos nodos se deben ingresar separados por un espacio (por ejemplo 0 3).

Luego de pedir todos los datos, convierte el string de matriz recibido a una matriz de numpy, esto lo hace con la funcion string_to_matriz, que transforma el string a listas de python con split, crea una matriz de numpy de tamaño nxn que contiene solo ceros, y después, viendo la matriz con listas se agregan los 1s a la matriz de numpy, y se retorna esta matriz.

Después se llama a la función do_tarea(m, k, par), que calcula el valor de m^k con la función recursiva pow(m, k) y luego retorna m_{ij}, siendo i y j los valores del par ingresado.

La función pow(m, k) hace lo siguiente: si k = 1, entonces retorna m (m^1), después calcula recursivamente la matriz tmp como tmp = pow(m, k/2), y después crea la matriz res como tmp*tmp (calcula la matriz como $m^{k/2} * m^{k/2}$), luego se analiza si k es par o impar, si k es par, entonces se retorna res, si es impar, entonces retorna res*m, esto es porque como se hace una división entera, con k impar res queda como m^{k-1}, por lo que se debe multiplicar por m para obtener m^k, y retorna ese valor

Ahora se calculara el orden del algoritmo con respecto al valor de k, en especifico de la función pow, ya que el buscar el elemento deseado en la matriz resultante toma tiempo constante.

Primero se puede ver que T(k) se calcula como T(k/2) (esto es al calcular recursivamente la matriz tmp), y luego multiplica dos matrices tmp para obtener res, y finalmente si k es impar se multiplica por m, las multiplicaciones entre matrices se tomarán como constante c, ya que no dependen del valor de k. Por lo que se obtiene la siguiente ecuación de recurrencia:

$$T(k) = T(k/2) + c$$

De la ecuación general del teorema maestro $T(n) = aT(\frac{n}{b}) + cn^d$, de la ecuación obtenida se tiene que $a = 1$, $b = 2$ y $d = 0$, y cae en el caso de $a = b^d$, ya que $1 = 2^0$, por lo que el algoritmo es de orden $k^d log(k)$, reemplazando por los valores de la ecuación, se tiene que el algoritmo es de orden $log(k)$.

2. P2

2.1. a

Primero se partirá demostrando que todo árbol es 2-coloreable.
Se tiene un árbol A = (V, E) con n vértices, por definición se cumple que es un grafo conexo y sin circuitos. Parto tomando un vértice cualquiera y lo pinto de color verde, ahora comienzo a analizar los vecinos de ese vértice, como no hay ningún par de vecinos unidos entre ellos (ya que no hay circuitos), puedo pintar a todos los vecinos de color rojo, y luego los vecinos de los vecinos se pintarían todos de color verde, esto es gracias a que no hay circuitos, y así se va intercalando entre rojo y verde hasta pintarlos a todos. Finalmente un árbol es 2-coloreable.

Ahora para demostrar que todo árbol es bipartito, se utiliza lo demostrado anteriormente de que un árbol es 2-coloreable, como los árboles son 2-coloreables, se pueden dejar todos los nodos de un color a un lado del grafo y los del otro color se dejan al otro lado, el grafo es bipartito ya que no hay conexiones entre nodos del mismo color, y para todo nodo en una de las particiones existe una arista que lo conecta con un nodo de la otra partición.

2.2. b

El lenguaje utilizado es Python 2.7, código adjunto en sección Anexos

El código desarrollado hace lo siguiente:

Se creó el objeto Tarea que contiene 2 variables de instancia: un diccionario colors donde se guardan los colores de cada grafo, y un Graph g.
El objeto Tarea contiene 3 métodos: get_root() que obtiene la raíz de el árbol self.g buscando el nodo que no tiene aristas apuntando hacia él (se asume que se entrega un árbol al comienzo). Otro método es pintar(v,c), que es un método recursivo que, si no ha sido pintado antes, pinta el vértice v del color c, y luego se llama a si misma para pintar los vecinos de v con el color contrario a c (esto colorea de 2 colores al grafo con colores 'a' y 'r', el color contrario a 'a' es 'r' y viceversa, no haber sido coloreado antes es tener color 'n'). El último método es do_tarea(n,s), donde n es el número de nodos de el árbol (enumerados de 1 a n) y s el el string que representa a los pares de aristas, en s cada par de aristas debe estar separado por \n, y en cada arista, los vértices están separados por un espacio (ej: n=5, s='1 2\n1 3\n2 4\n3 5').
El método do_tarea parte por transformar s a una lista, donde cada elemento es una arista, y una arista es una lista que contiene 2 vértices. Luego comienza a crear el grafo agregando los vértices de 1 a n, y a cada vértice se le asigna el color 'n' (sin color), después de esto agrega las conexiones entre los vértices y comienza a pintar el árbol obtenido. Para pintar se parte por obtener el nodo raíz del árbol con get_root(), y se llama a pintar(root, 'a'), que pinta la raíz de color 'a' y sus vecinos de color 'r', y continua pintando alternando los colores por cada nivel del árbol (el nivel es la distancia con la raiz, la raiz esta en 0, sus vecinos en 1 y los vecinos de los vecinos en 2, por lo que quedan todos los niveles pares y la raíz de color 'a' y todos los niveles impares de color 'r').
Ahora, como el árbol ya esta pintado, y como los árboles son bipartitos, se separan los vértices por color en listas va y vr, y empiezo a contar las aristas que puedo agregar como las conexiones que no hay entre va y vr. Finalmente se entrega la cuenta de las aristas.

3. Anexos

3.1. P1

```python
#!/usr/bin/env python
import numpy as np


def do_tarea(m, k, par):
    mk = pow(m, k)
    return int(mk[par[0]][par[1]])


def pow(m, k):
    if k == 1:
        return m
    tmp = pow(m, k/2)
    res = tmp.dot(tmp)
    if k % 2 == 0:
        return res
    else:
        return res.dot(m)


def string_to_matrix(s, n):
    # este if es por la forma en que pruebo el programa, si lo hago sin raw_input, entonces
    # el string contiene \n, si lo hago con raw_input, entonces contiene //n
    if '\n' in s:
        l = s.split('\n')
    else:
        l = s.split('\\n')
    m = []
    for i in range(0, len(l)):
        m.append(l[i].split(" "))
    res = np.zeros((n, n))
    for i in range(0, n):
        for j in range(0, n):
            if m[i][j] == '1':
                res[i][j] = 1
    return res


print "-----------------------------------------------------"
print "ingrese numero de nodos en el arbol (enumerados de 1 a n): "
print "si se presiona enter sin ingresar numero se elige n=5"
n = raw_input("n: ")
```

```
if n == '':
    n = 5
n = int(n)
print
print "\ningrese matriz de adyacencia"
print "si se presiona enter sin ingresar matriz se elige 0 0 1 0 1\\n0 0 1 0 0\\n1 1 0 1 0\\n0 0 1 0 0\\n1 0 0 0 0
m = raw_input("matriz: ")
if m == '':
    m = '0 0 1 0 1\n0 0 1 0 0\n1 1 0 1 0\n0 0 1 0 0\n1 0 0 0 0'

print
print "ingrese k: "
print "si se presiona enter sin ingresar k se elige k=2"
k = raw_input('k: ')
if k == '':
    k = 2
k = int(k)
print
print "ingrese par u, v: "
print "si se presiona enter sin ingresar par se elige (u, v) = 0 3"
par = raw_input('u v: ')
if par == '':
    par = '0 3'
p = par.split(' ')
p[0] = int(p[0])
p[1] = int(p[1])

matriz = string_to_matrix(m, n)
print
print 'respuesta: ' + str(do_tarea(matriz, k, p))
print "------------------------------------------------------"
```

3.2. P2

3.2.1. p2.py

```
#!/usr/bin/env python

from Graph import *


class Tarea:
    def __init__(self):
        self.colors = {}
        self.g = Graph()
```

```python
def do_tarea(self, n, s):
    # transformar string a matriz con listas
    R = string_to_matrix(s)
    n = int(n)
    # agregar vertices al grafo e inicializar colores en n
    for i in range(1, n + 1):
        ind = str(i)
        self.colors[ind] = 'n'
        self.g.add_vertex(ind)

    # agregar conexiones al grafo
    for t in R:
        self.g.add_edge(t[0], t[1])

    # pintar grafo con 2 colores: a y r
    root = self.get_root()
    self.pintar(root, 'a')

    va = []
    vr = []
    # como el arbol es bipartito, separo los nodos por color en 2 listas
    for v in self.g.get_vertexs():
        if self.colors[v] == 'a':
            va.append(v)
        else:
            vr.append(v)

    # convertir el grafo dirigido a grafo simple
    for v in self.g.get_vertexs():
        for e in self.g.get_edge(v):
            self.g.add_edge(e, v)

    # reviso las conexiones que faltan entre va y vr
    aristas = 0
    for v in va:
        e = self.g.get_edge(v)
        for ver in vr:
            if ver not in e:
                aristas += 1
    return aristas

def pintar(self, v, c):
    # pintar si no tiene ningun color
    if self.colors[v] == 'n':
        self.colors[v] = c
        for n in self.g.get_edge(v):
            self.pintar(n, next_color(c))
```

```
    def get_root(self):
        # obtiene la raiz del arbol buscando el nodo que no tiene aristas apuntando hacia el
        ver = self.g.get_vertexs()
        for v in self.g.get_vertexs():
            for e in self.g.get_edge(v):
                ver.remove(e)
        return ver[0]


def next_color(s):
    if s == 'a':
        return 'r'
    else:
        return 'a'


def string_to_matrix(s):
    # este if es por la forma en que pruebo el programa, si lo hago sin raw_input, entonces
    # el string contiene \n, si lo hago con raw_input, entonces contiene //n
    if '\n' in s:
        l = s.split('\n')
    else:
        l = s.split('\\n')
    m = []
    for i in range(0, len(l)):
        m.append(l[i].split(" "))
    return m


print "-----------------------------------------------------"
print "ingrese numero de nodos en el arbol (enumerados de 1 a n): "
print "si se presiona enter sin ingresar numero se elige n=5"
n = raw_input("n: ")
if n == '':
    n = 5
print
print "\ningrese pares de nodos que representan aristas"
print "si se presiona enter sin ingresar matriz se elige 1 2\\n1 3\\n2 4\\n3 5"
s = raw_input("matriz: ")
if s == '':
    s = '1 2\n1 3\n2 4\n3 5'

t = Tarea()
print t.do_tarea(n, s)
print "-----------------------------------------------------"
```

3.2.2. Graph.py

```python
#!/usr/bin/env python
# Graph implementation obtained from http://www.forosdelweb.com/f130/aporte-sencilla-implementacion-grafos-817941/


class Graph:
    # Simple graph implementation:
    # Directed graph
    # Without weight in the edges
    # Edges can be repeated (se modifico para que no se repitieran)

    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        # Add a vertex in the graph
        # Overwrite the value
        self.graph[vertex] = []

    def get_vertexs(self):
        # Get the vertexs in the graph
        return self.graph.keys()

    def del_vertex(self, vertex):
        # Remove the vertex if it's in the graph
        try:
            self.graph.pop(vertex)
        except KeyError:
            # Here vertex is not in graph
            pass

    def is_vertex(self, vertex):
        # Return True if vertex is in the graph
        # otherwise return False
        try:
            self.graph[vertex]
            return True
        except KeyError:
            return False

    def add_edge(self, vertex, edge):
        # Add a edge in vertex if vertex exists
        try:
            # grafo sin multiarcos
            if edge not in self.graph[vertex]:
                self.graph[vertex].append(edge)
```

```
        except KeyError:
            # Here vertex is no in graph
            pass

    def delete_edge(self, vertex, edge):
        # Remove a edge in vertex
        try:
            self.graph[vertex].remove(edge)
        except KeyError:
            # Here vertex is not in graph
            pass
        except ValueError:
            # Here the edge not exists
            pass

    def get_edge(self, vertex):
        # Return the edges of a vertex if the vertex is in the graph
        # Otherwise return None
        try:
            return self.graph[vertex]
        except KeyError:
            pass

    def __str__(self):
        # Print the vertex
        s = "Vertex -> Edges\n"
        for k, v in self.graph.iteritems():
            s += "%-6s -> %s\n" % (k, v)
        return s
```