Trabajo Práctico 2

Teoría de Algoritmos I (75.29)

Universidad de Buenos Aires

Facultad de Ingeniería

Segundo Cuatrimestre - 2016

Alumnos:

Emanuel Condo	(94773)
Martín Cura	(95874)
Fernando Nitz	(94994)
Nicolás Gago	(91677)

Programación dinámica

El problema de la mochila

Implementación

Se implementó una solución al problema de la mochila 0-1 con programación dinámica, basado en Kleinberg y Tardos (2006), capítulo 6.4, siguiendo un enfoque *profit* por el cual se calculan progresivamente los óptimos para los primeros elementos y con ellos una capacidad , a partir de los ya obtenidos, los cuales se van guardando en una matriz $n * w$. Completada la matriz, el valor óptimo buscado se encuentra en su última posición.

Esta resolución de programación dinámica guarda grandes cantidades de números (la matriz para los casos testeados tiene un tamaño en el orden de los millones de elementos) pero mediante esto evita repetir costosos cálculos recursivos. Para minimizar el costo en memoria, no se copian los grandes conjuntos involucrados, mas se acceden directamente.

Se incluye un parser para los archivos de prueba y una implementación , sin programación dinámica. Se asumió que lo que se debía comprobar era la obtención del valor óptimo, en vez del vector X que lo consigue.

Pruebas

Se utilizaron los archivos de prueba recomendados (de David Pisinger). *mochila.py* prueba secuencialmente cada *test* dentro de cada archivo en la subcarpeta especificada (con la entrega no se incluyen los archivos de prueba[1]).

Los *tests* corren en tiempos correspondientes con lo esperado: $O(NW)$, aunque cabe notar que no se comprobaron todos por el tiempo que involucraría. La última columna muestra cómo se mantiene esta correspondencia.

A continuación un extracto de corrida:[2]

[1] smallcoeff.tgz, 142 MiB; hardinstances.tgz, 45 MiB.
[2] Datos de corrida: Dell 15z con Ubuntu 16.04, Intel Core i7-2640M CPU @ 2.8 - 3.5 GHz y 8 GB de RAM.

```
tp2/Mochila$ python mochila.py
--------------
knapPI_15_200_1000.csv

knapPI_15_200_1000_1  N*W = 200598     0.157191 s   (vs 0.0 s)     N*W / t = 1276141.76384
knapPI_15_200_1000_2  N*W = 401799     0.210521 s   (vs 0.0 s)     N*W / t = 1908593.44198
knapPI_15_200_1000_3  N*W = 577674     0.293233 s   (vs 0.0 s)     N*W / t = 1970017.01718
knapPI_15_200_1000_4  N*W = 848019     0.43986 s    (vs 0.0 s)     N*W / t = 1927929.34115
knapPI_15_200_1000_5  N*W = 1008216    0.527791 s   (vs 0.0 s)     N*W / t = 1910256.14306
knapPI_15_200_1000_6  N*W = 1239366    0.658807 s   (vs 0.0 s)     N*W / t = 1881227.73437
knapPI_15_200_1000_7  N*W = 1406196    0.749978 s   (vs 0.0 s)     N*W / t = 1874982.9995
knapPI_15_200_1000_8  N*W = 1644783    0.879222 s   (vs 0.0 s)     N*W / t = 1870725.4823
knapPI_15_200_1000_9  N*W = 1598151    0.85721 s    (vs 0.0 s)     N*W / t = 1864363.4582
knapPI_15_200_1000_10 N*W = 2100249    1.138937 s   (vs 0.0 s)     N*W / t = 1844043.17359
knapPI_15_200_1000_11 N*W = 2223663    1.204669 s   (vs 0.0 s)     N*W / t = 1845870.52543
knapPI_15_200_1000_12 N*W = 2357127    1.270597 s   (vs 0.01 s)    N*W / t = 1855133.45301
knapPI_15_200_1000_13 N*W = 2556117    1.398722 s   (vs 0.0 s)     N*W / t = 1827466.0726
knapPI_15_200_1000_14 N*W = 2781036    1.507621 s   (vs 0.0 s)     N*W / t = 1844651.93839
knapPI_15_200_1000_15 N*W = 3194694    1.774671 s   (vs 0.0 s)     N*W / t = 1800161.26933
knapPI_15_200_1000_16 N*W = 3209769    1.743507 s   (vs 0.0 s)     N*W / t = 1840984.292
knapPI_15_200_1000_17 N*W = 3117108    1.693991 s   (vs 0.0 s)     N*W / t = 1840097.14337
knapPI_15_200_1000_18 N*W = 3541218    1.934385 s   (vs 0.0 s)     N*W / t = 1830668.66213
knapPI_15_200_1000_19 N*W = 3821211    2.079325 s   (vs 0.0 s)     N*W / t = 1837717.0476
knapPI_15_200_1000_20 N*W = 4161102    2.27137 s    (vs 0.0 s)     N*W / t = 1831978.93782

--- passed ---
```

Código

mochila.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
###                          ###
###  Problema de la mochila  ###
###                          ###
import sys
import csv
import time
from os import listdir
from os.path import isfile, join

# a: array, p: profit, v: value, w: weight

# Sin programación dinámica
def P_dull(i, w):
    if i < 0:
        return 0
    if aw[i] > w:
        return P_dull(i-1,w)
    return max( P_dull(i-1, w),  P_dull(i-1, w-aw[i]) + av[i] )

def knapsack_dull(N, W):
    return P_dull(N, W)


def knapsack(N, W):
    M = [[0 for p in xrange(W+1)] for p in xrange(N+1)]
```

```
    def P(i, w):
        if i <= 0 or w == 0:
            return 0
        if aw[i] > w:
            return M[i-1][w]
        return max( M[i-1][w], M[i-1][w-aw[i]] + av[i] )

    for i in xrange(1, N+1):
        for w in xrange(1, W+1):
            M[i][w] = P(i, w)
    return M[N][W]


#dirname = "smallcoeff_pisinger"
dirname = "hardinstances_pisinger"

files = [f for f in listdir(dirname) if isfile(join(dirname, f))]
for filename in files:
    if ".csv" not in filename:
       continue

    with open(join(dirname, filename)) as csvfile:
        print "--------------"
        print filename
        print
        reader = csv.reader(csvfile, delimiter=' ')
        seguir = True

        try:
            while (seguir):
                test_name = reader.next()[0]
                N = int(reader.next()[1])
                W = int(reader.next()[1])
                z = int(reader.next()[1])
                t = float(reader.next()[1])
                av = [0]
                aw = [0]
                xl = [0]
                for row in reader:
                    if "--" in row[0]:
                        break
                    (i, vi, wi, xi) = map(int, row[0].split(','))
                    av.append(vi)
                    aw.append(wi)
                    xl.append(xi)
                reader.next()  # Ignora línea vacía tras cada test

                print test_name, '\t', "N*W =", (N+1) * (W+1), "\t ",
                sys.stdout.flush()
                start = time.clock()

                k = knapsack(N, W)

                elapsed = time.clock() - start

                if k != z:
                    print
                    print k, "!=", z, "!!!"
                    assert k == z

                print elapsed, "s\t(vs", t, "s)\t",
                print "N*W / t =", (N+1) * (W+1) / elapsed

        except StopIteration:
            print "--- passed ---"
            print
```

El problema del viajante de comercio

Implementación

Se desarrolló la implementación del problema del viajante utilizando programación dinámica (basado en el algoritmo de Bellman-Held-Karp), en el cual se utilizó un enfoque Top-Down, es decir, se dividió al problema del viajante en subproblemas y estos se resuelven "recordando" las soluciones (las soluciones son almacenadas en una estructura de tipo hash) por si fueran necesarias nuevamente. Utiliza una combinación de *Memoization* [3] y recursividad.

Para tratar de resolver este tipo de problemas de manera exacta, su solución más directa podría ser, intentar todas las permutaciones posibles y ver cuál de esas es la de menor costo (Búsqueda por fuerza bruta). Sin embargo, el tiempo de ejecución es de orden polinomial $O(n!)$ con lo cual es impracticable (tomando un n > 20, el tiempo de ejecución ronda desde días hasta miles de años).

Con el algoritmo de Bellman-Held-Karp, la complejidad es O ($n^2 . 2^n$).

El problema del viajante se puede pensar como un conjunto de nodos conectados entre sí por aristas, lo que conocemos como Grafo. Entonces, el grafo puede ser dirigido o no dirigido y utilizando el algoritmo de Bellman-Held-Karp, se pueden obtener soluciones óptimas a subproblemas del problema original pero por otro lado se realizarán cálculos repetidos, por eso con la programación dinámica se pretende guardar esas soluciones aumentando aún más la eficiencia del algoritmo, ya que se ahorraría de cálculos innecesarios.

A continuación se muestra el algoritmo de recurrencia para el ciclo hamiltoniano:

$$D(v,S) = \begin{cases} \text{si } S = \emptyset: & c_{vv_0} \\ \text{si no}: & \min\limits_{u \in S} [c_{vu} + D(u, S - \{u\})] \end{cases}$$

Pruebas

Para los casos de prueba se utilizaron los siguientes archivos:

- p01.tsp (matriz asimétrica)
- testfile_18_symmetric.tsp

[3] Memorization : https://en.wikipedia.org/wiki/Memoization

- testfile_18_asymmetric.tsp
- testfile_20_symmetric.tsp
- testfile_20_asymmetric.tsp

Obteniendo los siguientes resultados de tiempos de ejecución[4]:

Test Files	Tiempo de ejecución (seg)
p01.tsp	3.18601298332
testfile_18_symmetric.tsp	47.9638359547
testfile_18_asymmetric.tsp	38.5003070831
testfile_20_symmetric.tsp	246.245939016
testfile_20_asymmetric.tsp	224.157764912
testfile_22_symmetric.tsp	588.986454964
testfile_21_asymmetric.tsp	512.479609013

Conclusiones:

Mediante varias pruebas, podemos corroborar que el tiempo para obtener la solución al problema del viajante, incrementa de manera notoria al aumentar la cantidad de ciudades (con n mayor a 21, los tiempos de ejecución son, computacionalmente, altos). Sin embargo, para valores de n menores a 22 se obtuvieron resultados aceptables. Podemos concluir que la programación dinámica nos ayuda a resolver problemas de manera eficiente en el cual se pueden utilizar distintos enfoques (Top-Down / Bottom-Up), utilizando las soluciones óptimas obtenidas de los conjuntos de subproblemas del problema original.

Código

main.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
###   Problema del viajante TSP    ###
import sys
```

[4] Datos de corrida: Acer S3 con Ubuntu 14.04, Intel Core i5-3317U CPU @ 1.7 - 2.4 GHz y 4 GB de RAM.

```
from parser import *
from viajante import *

cant_args = len(sys.argv) - 1
if cant_args != 2:
        print 'Cantidad de argumentos inválidos. Ejecute el siguiente comando:'
        print '> python main.py <vértice> testfiles/<archivo_de_prueba>'
        quit()

testfile = str(sys.argv[2])
if not '.tsp' in testfile:
        print 'Archivo de prueba en formato inválido.'
        quit()

if str(sys.argv[1]).isdigit() == False:
        print 'Vértice inválido. Ingrese un número no negativo.'
        quit()

vertice_id = int(sys.argv[1])

M = parse(testfile)
viajante(vertice_id, M)
```

viajante.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*- #
From time import time

# 0 < inicio <= len(M)
def viajante(inicio, M):
        S = list(range(len(M)))
        v = inicio - 1
        S.remove(v)
        costos = {}
        parents = {}
        rec = []

        def D(x, conjunto):
                if (str(x)+str(conjunto)) in  costos:
                        return costos[str(x)+str(conjunto)]

                # Para matrices simétricas y asimétricas
                if (len(conjunto) == 0):
                        try:
                                return M[x][v]
                        except IndexError:
                                return M[v][x]
```

```
                values = []
                for elem in conjunto:
                subconjunto = list(conjunto)
                subconjunto.remove(elem)
                try:
                        values.append(M[x][elem] + D(elem, subconjunto))
                except IndexError:
                        values.append(M[elem][x] + D(elem, subconjunto))
                costo_min = min(values)
                costos[str(x)+str(conjunto)] = costo_min
                prox_vert_cercano = conjunto[values.index(costo_min)]
                parents[str(x)+str(conjunto)] = prox_vert_cercano
                return min(values)

        def recorrido(x, conjunto):
                if len(conjunto) == 0:
                        return
                vertice = parents[str(x)+str(conjunto)]
                rec.insert(0, vertice+1)
                subconjunto = list(conjunto)
                subconjunto.remove(vertice)
                recorrido(vertice, subconjunto)

        t_start = time()
        costoTotal = D(v, S)
        time_diff = time() - t_start
        print 'Tiempo de ejecución: ', time_diff, ' segundos'
        recorrido(v, S)
        rec.insert(0, v+1)
        rec.append(v+1)
        print 'Recorrido: ', rec
        print 'Costo Total:', costoTotal
```

parser.py

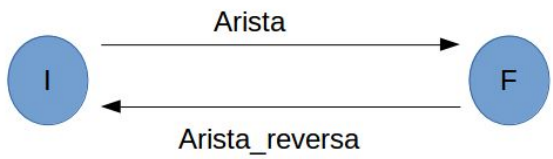
```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import csv

'''
Secciones de los archivos de ejemplo
'NAME'
'TYPE'
'DIMENSION'
'EDGE_WEIGHT_TYPE'
'EDGE_WEIGHT_FORMAT'
```

```
'NODE_COORD_TYPE'
'DISPLAY_DATA_TYPE'
'TOUR_SECTION'
'''

# Para probar att48_d.txt, metelo en un archivo con DIMENSION
# y EDGE_WEIGHT_FORMAT setteados, la 2da a FULL_MATRIX, y la
# etiqueta EDGE_WEIGHT_SECTION antes de la matriz.

def parse(filename):
        dicc = {}
        tour_section = []
        matriz = []

        with open(filename) as csvfile:
                reader = csv.reader(csvfile, delimiter=' ')
                print 'Parseando archivo ', filename
                for linea in reader:
                        #print linea #
                        if (linea[0] == 'EOF'):
                                break
                        elif (linea[0] == 'TOUR_SECTION'):
                                for i in xrange(int(dicc['DIMENSION']) + 2): #ema revisar
                                        linea = reader.next()
                                        tour_section.append(int(linea[-1]))
                        elif (linea[0] == 'EDGE_WEIGHT_SECTION'):
                                matriz = []
                                if (dicc['EDGE_WEIGHT_FORMAT'] == 'FULL_MATRIX'):
                                        for i in xrange(int(dicc['DIMENSION'])):
                                                linea = reader.next()
                                                matriz.append(map(int, filter(lambda x: x != '', linea)))
                                elif (dicc['EDGE_WEIGHT_FORMAT'] == 'LOWER_DIAG_ROW'):
                                        for i_fila in xrange(int(dicc['DIMENSION'])):
                                                fila = []
                                                for i_col in xrange(i_fila + 1):
                                                linea = reader.next()
                                                fila.append(int(linea[0]))
                                                #for aux in xrange(i_fila + 1, int(dicc['DIMENSION'])):
                                                #       fila.append(0)
                                                matriz.append(fila)
                                        # Faltaría espejar la matriz si es necesario
                        else:
                                dicc[linea[0].split(":")[0]] = " ".join(
                                                        filter(lambda x: x != ':', linea[1::]))
                print 'Fin de parseo del archivo ', filename
        return matriz
```

Flujo de redes

Implementación

Para la resolución del problema se modeló el mismo mediante un flujo de red, basándose en Kleinberg y Tardos (2006), capítulo 7.11, en el cual se puede ver una versión algo más compleja, ya que en esta última tenemos proyectos que dependen de proyectos, independientemente de que estos den ganancia o pérdida, teniendo el grafo con un nivel máximo que depende de las dependencias entre los proyectos.
En cambio, el grafo que tendremos para resolver el problema en cuestión tendrá un nivel máximo de valor 4, yendo de S, pasando luego por algún proyecto, luego por alguna de las áreas necesitadas para el proyecto, y finalmente llegando a T.
Para el modelado del grafo es importante remarcar que, a pesar de ser un grafo dirigido, se agregan al grafo las reversas de las aristas, las cuales son indispensables para obtener el minimal cut, dado que si no no podríamos incluir proyectos cuya arista entrante se encuentre saturada.
En otras palabras, pasamos de tener una arista dirigida a tener dos aristas que viajan en sentidos contrarios.



La ilustración representa el cambio que se realizó en el gráfico para poder soportar los cambios necesarios para la segunda parte del tp.

La implementación se puede resumir en 4 etapas:

- El procesamiento del archivo de entrada.
- La inicialización del grafo, el cual es el mismo del TP1 , con las modificaciones correspondientes para cumplir lo antedicho.
- La obtención del flujo máximo, que en este caso se decidió realizar mediante el algoritmo de Ford-Fulkerson (se podrían haber utilizado otros, como por ejemplo el de Edmonds-Karp, el cual es una implementación más eficiente del primero).
- La obtención del Minimal Cut, que se realizará mediante un Depth First Search en el grafo residual.

Complejidad

El cálculo de la complejidad del algoritmo se determinará por la suma de las complejidades de las últimas 2 etapas, es decir:

COMPLEJIDAD(algoritmo) = COMPLEJIDAD(ford-fulkerson) + COMPLEJIDAD(dfs)

Siendo M el número de proyectos, N el número de áreas y R el número de requisitos por parte de todos los proyectos

Complejidad del Ford-Fulkerson:
Como se puede notar en el código,

```
def fordFulkerson(g, src, dst):
    path = g.find_path(src, dst, [])
    while path != None:
        residuals = [arista.weight - g.flow[arista] for arista in path]
        flow = min(residuals)
        for arista in path:
            g.flow[arista] += flow
            g.flow[arista.reversa_arista] -= flow
        path = g.find_path(src, dst, [])
    print g.flow
    return sum(g.flow[arista] for arista in g.get_A_Adj(src))
```

El algoritmo cicla sobre las aristas y aumentando los flujos.
Traduciendo lo dicho anteriormente, para el código armado, se comenzará con flujo 0 a recorrer desde el nodo inicio hasta el nodo final ciclando una cantidad de caminos posibles entre ambos.
Sabiendo que A es la cantidad de aristas en el grafo, y F el flujo máximo obtenido, podemos deducir que la complejidad del algoritmo es O(A.F).
Debemos notar 2 detalles:

1) El ciclo anteriormente mostrado recorre por path y no por aristas, sin embargo el mismo es composición de las aristas y por ende suponiendo que se repetirán un número constante máximo de veces, podrá quedar y ser representado por A.
2) El flujo aumentará en cada iteración y empezará en 0. Al no poder especular más dicha cantidad, más que saber que siempre incrementará en mínimo una unidad, concluimos que esta parte es O(F)

Finalmente uniendo ambos detalles, justificamos el orden anteriormente mencionado (O(A.F)).

Complejidad del DFS
Como se puede notar también en el código,

```
def minimalCut(self, src):
    visited = set()
    nodosARecorrer = [src]
    resultado = []
```

```
    while nodosARecorrer:
        nodo = nodosARecorrer.pop()
        if nodo not in visited:
            visited.add(nodo)
            for arista in self.get_A_Adj(nodo):
                destino = arista.dst
                if destino not in visited and arista.weight - self.flow[arista] <>0 :
                    resultado.append(destino)
                    nodosARecorrer.append(destino)

    return resultado
```

en este caso, es más sencillo el análisis de complejidad del algoritmo. Esto se debe a que el algoritmo busca los vértices y de cada uno de ellos, recorre las aristas con las que puede alcanzar a sus vecinos.
De esta forma, continuando la convención de llamar A a la cantidad de aristas y V la cantidad de vértices, concluimos que la complejidad de minimal cut, implementado con un bfs es de O(V.A).

Finalmente y como conclusión del análisis de complejidad del algoritmo, podemos decir que la complejidad total es: O(A.F) + O(V.A).

Traduciendo lo obtenido a la nomenclatura pedida por el enunciado, en donde, m es el número de proyectos, n el número de áreas, y r la cantidad total de requisitos, podemos concluir que:

1) V = m + n + 2
2) A = m + n + r
3) F = r (esto sabiendo que F es aproximadamente igual a la cantidad de paths y por lo tanto, la cantidad de requisitos, que son los que conectan a los especialistas y a los proyectos)

Dando como resultado, $O(r.m + r.n + r^2)) + O (m^2 + n.m + m + n^2 + n + r.m + r.n + r)$ y por lo tanto, $O(n^2 + r^2 + m^2 + r.m + r.n + n.m)$.

Dado que r > m y r > n. Entonces finalizamos concluyendo que nuestro algoritmo se encuentra en $O(r^2)$.

Código:

main.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from fdr import *

procesoArchivo("archivo2.txt")
grafo = inicializoGrafo()
print "flujo maximo = " + str(grafo.fordFulkerson(0,grafo.V()-1))
print "corte minimo = " + str(grafo.minimalCut(0))
```

Fdr.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from proyecto import *
from especialista import *
from tadgrafo2 import *

def procesoArchivo(archivo):
	# SE LEE EL ARCHIVO DE CONFIGURACION INICIAL Y SE DEJA INICIALIZADAS LAS
VARIABLES
	#
	# n: se deja la cantidad de areas o especialidades
	# m: son la cantidad de proyectos
	# costo_areas: es un diccionario que tiene como id numero de area y valor
su especialista
	# ganancia_req: es un diccionario que tiene como id numero de proy y
valor su proyecto
	archivo = open(archivo)
	n = archivo.readline()
	n = n.rstrip('\n')
	n = int(n)
	print n
	m = archivo.readline()
	m = m.rstrip('\n')
	m = int(m)
	print m
	global costo_areas
	costo_areas = []
	global ganancia_req
	ganancia_req = []

	linea = archivo.readline()
	i = 0
	while i < n:
		numeros = linea.split(" ")
		nuevaLinea = []
		for elem in numeros:
			nuevaLinea.append(int(elem.rstrip('\n')))
		especialista = Especialista(nuevaLinea.pop())
		costo_areas.append(especialista)
		linea = archivo.readline()
		i = i + 1
```

```
	i = 0
	while i < m:
		numeros = linea.split(" ")
		nuevaLinea = []
		for elem in numeros:
			nuevaLinea.append(int(elem.rstrip('\n')))
		proyecto = Proyecto(nuevaLinea.pop(0),nuevaLinea)
		ganancia_req.append(proyecto)
		linea = archivo.readline()
		i = i + 1

	archivo.close()

def inicializoGrafo():
#el tadgrafo se toma como un modulo cerrado, dada su implementación, y debido a
la necesidad de utilizar nombres para
# esclarecer el asunto, se creara un diccionario nodos, con clave = nombre y
valor = numeroDeNodo
	grafo = Digraph(2+len(costo_areas)+len(ganancia_req))
	global nodos
	nodos = {}
	nodos["s"] = 0
	nodos["t"] = grafo.V()-1
	for i in range(0,len(ganancia_req)):
		nodos["proyecto"+ str(i+1)] = i+1
		grafo.add_edge( 0, i+1, ganancia_req[i].get_ganancia())
	for i in range(0,len(costo_areas)):
		numeroDeNodo = i+len(ganancia_req)+1
		nodos["especialista" + str(i+1)] = numeroDeNodo
		grafo.add_edge( numeroDeNodo, nodos["t"],
costo_areas[i].get_sueldo_especialista())
	#creo las aristas entre los proyectos y los especialistas
	gananciaMaxima = 0
	for i in ganancia_req:
		gananciaMaxima = gananciaMaxima + i.get_ganancia()

	for i in range(0,len(ganancia_req)):
		areas_requeridas = ganancia_req[i].get_areas_requeridas()
		for j in range(0,len(areas_requeridas)):
			#agrego la gananciaMaxima + 1 para simular infinito
			proyecto = str("proyecto") + str(i+1)
			especialista = "especialista"+ str(areas_requeridas[j])
			peso = gananciaMaxima +1
			grafo.add_edge(nodos[proyecto], nodos[especialista], peso)

	return grafo
```

Tadgrafo2.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import random
from tadCaminos import *
from constantes import *
from collections import defaultdict

class Vert:
    def __init__(self,node):
        self.id = node
        self.vecinos = {}
        self.visitado = False
        self.distancia = POSITIVE_INFINITY
        self.padre = None

    def add_neighbor (self ,dst,w = 0):
        self.vecinos[dst] = w

    def get_neighbors (self):
        return self.vecinos

    def get_id(self):
        return self.id

    def get_weight (self, dst):
        return self.vecinos[dst].get_weight()

    def visitar(self):
        self.visitado = True

    def fue_visitado(self):
        return self.visitado

    def setear_no_visitado(self):
        self.visitado = False

    def set_padre(self, padre):
        self.padre = padre

    def set_distancia(self, distancia):
        self.distancia = distancia

    def set_distancia_inf(self):
```

```
        self.distancia = POSITIVE_INFINITY

    def set_padre_vacio(self):
        self.padre = None

    def get_distancia(self):
        return self.distancia

    def get_padre(self):
        return self.padre

class Arista:
    """Arista de un grafo.
      """
    def __init__(self, src, dst, weight=0):
        # inicializar y do things
        self.src = src
        self.dst = dst
        self.weight = weight

    def get_weight (self):
        return self.weight

    def get_from (self):
        return self.src

    def get_to (self):
        return self.dst

    def __repr__(self):
        return "%s->%s:%s" % (self.src, self.dst, self.weight)

class Digraph:
    """Grafo no dirigido con un número fijo de vértices.
    Los vértices son siempre números enteros no negativos. El primer vértice
    es 0.

    El grafo se crea vacío, se añaden las aristas con add_edge(). Una vez
    creadas, las aristas no se pueden eliminar, pero siempre se puede añadir
    nuevas aristas.
    """

    def __init__(g, V):  # @NoSelf
        """Construye un grafo sin aristas de V vértices.
        """
        g.aristas = []
```

```
        g.vertices = {}

        g.adj = defaultdict(list)
        g.flow = {}

        for i in range(0,V):
            new_vert = Vert(i)
            g.vertices[i] = new_vert

    def V(g):   # @NoSelf
        """Número de vértices en el grafo.
        """
        return len(g.vertices)

    def E(g):   # @NoSelf
        """Número de aristas en el grafo.
        """
        return len(g.aristas)

    def adj_e(g, v):    # @NoSelf
        """Itera sobre los aristas incidentes _desde_ v.
        """
        aristas_incidentes_de_v = []
        for i in range(0,len(g.aristas)):
            if g.aristas[i].get_from() == v :
                aristas_incidentes_de_v.append(g.aristas[i])
        return aristas_incidentes_de_v



    def adj(g, v):  # @NoSelf
        """Itera sobre los vértices adyacentes a 'v'.
        """
        return g.vertices[v].get_neighbors().keys()


    def add_edge(g, u, v, weight=0):    # @NoSelf
        """Añade una arista al grafo.
        """
        g.vertices[u].add_neighbor(g.vertices[v],weight)
        new_edge = Arista(u,v,weight)
        g.aristas.append(new_edge)

        arista = Arista(u,v,weight)
        reversa_arista = Arista(v,u,0)
        arista.reversa_arista = reversa_arista
```

```
        reversa_arista.reversa_arista = arista
        g.adj[u].append(arista)
        g.adj[v].append(reversa_arista)
        g.flow[arista] = 0
        g.flow[reversa_arista] = 0


    def __iter__(g):    # @NoSelf
        """Itera de 0 a V."""
        return g.vertices

    def get_V(g,v): # @NoSelf
        return g.vertices[v]

    def get_A(self, src, dst):
        for i in range(0,len(self.aristas)):
            arista = self.aristas[i]
            if (arista.get_from() == src ) and (arista.get_to() == dst) :
                return arista
        return None

    def get_A_Adj(g, v):
        return g.adj[v]

    def iter_edges(g):  # @NoSelf
        """Itera sobre todas las aristas del grafo.

        Las aristas devueltas tienen los siguientes atributos de solo lectura:

            • e.src
            • e.dst
            • e.weight
        """
        return iter(g.aristas)

    def sacar_visitados(g): # @NoSelf
        for i in range(0,len(g.vertices)):
            g.vertices[i].setear_no_visitado()

    def poner_distancias_inf(g): # @NoSelf
        for i in range(0,len(g.vertices)):
            g.vertices[i].set_distancia_inf()

    def eliminar_padres(g): # @NoSelf
        for i in range(0,len(g.vertices)):
            g.vertices[i].set_padre_vacio()
```

```
    def find_path(g, src, dst, path):
        if src == dst:
            return path
        for arista in g.get_A_Adj(src):
            residual = arista.weight - g.flow[arista]
            if residual > 0 and arista not in path:
                result = g.find_path( arista.dst, dst, path + [arista])
                if result != None:
                    return result

    def fordFulkerson(g, src, dst):
        path = g.find_path(src, dst, [])
        while path != None:
            residuals = [arista.weight - g.flow[arista] for arista in path]
            flow = min(residuals)
            for arista in path:
                g.flow[arista] += flow
                g.flow[arista.reversa_arista] -= flow
            path = g.find_path(src, dst, [])
        print g.flow
        return sum(g.flow[arista] for arista in g.get_A_Adj(src))

    def minimalCut(self, src):
        visited = set()
        nodosARecorrer = [src]
        resultado = []
        while nodosARecorrer:
            nodo = nodosARecorrer.pop()
            if nodo not in visited:
                visited.add(nodo)
                for arista in self.get_A_Adj(nodo):
                    destino = arista.dst
                    if destino not in visited and arista.weight -
self.flow[arista] <>0 :
                        resultado.append(destino)
                        nodosARecorrer.append(destino)

        return resultado
```

Constantes.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
POSITIVE_INFINITY = float("inf")
```

Especialista.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

class Especialista:
    def __init__(self, sueldo):
        self.costo = sueldo
        self.fue_contratado = False

    def get_sueldo_especialista(self):
        return self.costo

    def contratar(self):
        self.fue_contratado = True

    def no_contratar(self):
        self.fue_contratado = False

    def tiene_trabajo(self):
        return self.fue_contratado
```

Proyecto.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

class Proyecto:
    def __init__(self, ganancia,areas_requeridas):
        self.ganancia = ganancia
        self.areas_requeridas = areas_requeridas
        self.realizar_proyecto = False

    def contratar (self):
        self.realizar_proyecto = True

    def no_contratar(self):
        self.realizar_proyecto = False

    def fue_contratado(self):
        return self.realizar_proyecto

    def get_ganancia(self):
        return self.ganancia

    def get_areas_requeridas(self):
        return self.areas_requeridas
```

Archivos de prueba:

archivo.txt

```
3
2
16
13
9
20 1 2
10 3
```

archivo2.txt

```
1
2
15
10 1
20 1
```