Trabajo Práctico N°3

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

23 de junio de 2017

Índice

1. Programación Dinámica

El problema propuesto es similar al de encontrar el mínimo y máximo elemento en un array de números con la restricción de que el mínimo esté antes que el máximo.

El algoritmo parte de la base de que se compra y se vende el primer día (Es decir, se empieza con una ganancia igual a 0), y guarda el *mínimo actual*. Avanzando en los días, si vendiendo en el día en el que se está da un beneficio mayor a lo que ya se compró y vendió, cambia y pasa a vender en este día y a comprar en el del *mínimo actual*; si no es así, sigue. Si el precio del día actual es menor al *mínimo actual*, entonces es un potencial día de compra y se actualiza el *mínimo actual*.

La ecuación de recurrencia sería algo así:

$$S_{i+1} = \{max\{S_i, k[i+1] - minActual_i\}\} \tag{1.1}$$

Donde:

- En cada S_i se guarda el mínimo y máximo que dan esta diferencia (siendo S_i la diferencia máxima).
- k es el array con los valores de compra/venta.

1.1. Cómo correrlo

Para correr el algoritmo basta con importar la función `compraVenta` del archivo `pg.py`, y utilizarla pasándole un array con los números. Dentro del archivo `pg_test.py` se puede ver un ejemplo de esto, donde se verifica que el resultado sea correcto comparándolo con el resultado de un algoritmo cuadrático.

2. Algoritmos Randomizados

La idea del algoritmo de contracción de Karger es encontrar el corte mínimo en un grafo $G = (V, E)$. Esto es, dos conjuntos no vacíos A y B, donde $A \cap B = \varnothing$ y $A \cup B = V$. El tamaño del corte (A, B) se define como el número de aristas $e = (u, v)$ con $u \in A$ y $v \in B$, o viceversa y lo que se busca es que éste sea mínimo.

Este algoritmo es un caso de randomización de tipo **Monte Carlo**, ya que tiene una probabilidad de que el mismo falle en encontrar la solución correcta.

El algoritmo de contracción en sí consiste en elegir una arista $e = (u, v)$ al azar, y "fusionar" los vértices creando un *supernodo*, el cual tiene tiene todas las aristas de u y v (Salvo la/s arista/s entre ellos). Es importante recalcar que el grafo debe permitir múltiples aristas entre dos vértices, ya que al fusionar es importante que se mantengan la cantidad de aristas (Nuevamente, salvo las que son entre u y v). Este proceso se debe repetir hasta que el grafo sólo conste de 2 vértices (Notar que en cada "iteración" eliminamos un nodo), los cuales se corresponderán con el corte (A, B), y la cantidad de aristas en el grafo será el tamaño del mismo.

Lo más interesante de esto es que su probabilidad de falla es relativamente alta, pero puede ser reducida ampliamente mediante múltiples (en cantidad polinomial) corridas. Yendo a los números, la probabilidad de que el algoritmo sea correcto con una única corrida es de al menos $\binom{n}{2}^{-1}$ (con $n = |V|$), lo cual para un n grande es un número muy chico (Es decir, su probabilidad de falla es como mucho $1 - \binom{n}{2}^{-1}$). Pero si el mismo se corre $\binom{n}{2}$ veces, se reduce a que se puede fallar en encontrar el corte mínimo con una probabilidad:

$$(1 - \binom{n}{2}^{-1})^{\binom{n}{2}} \leq \frac{1}{e} \tag{2.1}$$

Aún más, si se corre $\binom{n}{2} * \ln n$ veces, la probabilidad en 2.1 desciende a $e^{-\ln n} = \frac{1}{n}$, lo cual es más que aceptable para un n grande.

El problema de esta solución es que su tiempo de corrida es $O(n^4 * \log n)$. Es por ello que se plantea una alternativa un tanto más rápida, que corre en $O(n^2 * \log^3 n)$. Ésta se basa en que a medida que el grafo se achica, la probabilidad de contraer una arista que pertenece al corte mínimo aumenta. Para solucionar esto, el algoritmo se corre en 2 fases: la *segura* y la *peligrosa*. En la primera se contrae el grafo pasando de n vértices a $n/\sqrt{2} + 1$, donde la probabilidad de que ninguna de estas contracciones toque el corte mínimo es de al menos $\frac{1}{2}$. La segunda fase se corre 2 veces y de forma recursiva (Hasta que el grafo tenga 8 vértices o menos, cuando pasa a usarse *fuerza bruta*), para obtener 2 resultados distintos y quedarse con el de menor valor. De esta manera, la probabilidad de que este algoritmo devuelva el resultado correcto es:

$$P(n) \geq 1 - (1 - \frac{1}{2}P(\frac{n}{\sqrt{2}} + 1))^2 \tag{2.2}$$

Donde en 2.2, $P(n) = \Omega(1/\log n) \geq \alpha/\ln n$, para $\alpha > 2$. Y nuevamente, si se corre múltiples veces, por ejemplo $c * \ln^2 n$ (Siendo c constante), la probabilidad de éxito es al menos $1 - \frac{1}{n^{c/\alpha}}$, con lo que modificando c se puede cambiar esta probabilidad a un número arbitrariamente grande.

2.1. Cómo correrlo

Correr el algoritmo con una instancia del grafo con n vértices y $2*n$ aristas es tan simple como correr `python karger.py n` desde la carpeta `src` del proyecto.

3. Algoritmos Aproximados

El objetivo del algoritmo es resolver el problema de optimización de Subset Sum: dado un set de enteros positivos y t un entero positivo, encontrar la suma más grande de enteros del set que sea menor a t.

La idea del algoritmo aproximado de tiempo polinomial viene del algoritmo que lo resuelve de manera exacta, que lo hace en tiempo exponencial. Este mismo surge de una suerte de fuerza bruta, es decir, de calcular la suma de todos los subsets y luego elegir el más cercano a t. Pero el algoritmo exacto lo hace de manera más inteligente, ya que para calcular la suma de todos los subsets $x_1, ..., x_i$, utiliza la suma de todos los subsets $x_1, ..., x_{i-1}$, y también si se da cuenta que la suma de un cierto subset da mas que t, lo elimina. En resumen, el algoritmo exacto toma un subset $S = x_1, ..., x_n$ y el entero t, luego va calculando L_i que es la lista de todos los subsets de $x_1, ..., x_i$ que no superan t, y termina devolviendo el valor más alto de L_n.

Este algoritmo es exacto pero muy costoso. Buscando una solución lo suficientemente buena y menos costosa podemos usar el aproximado de tiempo polinomial. Este algoritmo ataca el problema de que calcular L_i en cada paso es muy costoso, y lo resuelve recortando (*trimming*) cada lista L luego de crearla. Usa la idea de que si hay 2 valores muy cercanos en la lista no vale la pena mantenerlos a ambos. Utilizando un parámetro δ de recorte, recorta todos los elementos y tal que existe z en la L' recortada de manera que:

$$\frac{y}{1+\delta} \leq z \leq y \tag{3.1}$$

Este método se basa en que se pueden quitar muchos elementos sin que ellos queden sin ser representados en la lista recortada. El metodo de recortar la lista L en $\Theta(|L|)$, es usando la lista ordenada, donde agarra el primer elemento y luego va agregando los elementos más grandes mientras no cumpla 3.1. Finalmente, la lista es L'.

En resumen, el algoritmo polinómico funciona igual que el exacto pero utiliza el algoritmo de recorte luego de calcular L_i, que se calcula utilizando un parámetro de aproximación $0 < \epsilon < 1$ que se usa para calcular $\delta = \dfrac{\epsilon}{2n}$ siendo $n = |S|$.

3.1. Cómo correrlo

Se puede crear un problema aleatorio con el método `generar_problema_aleatorio` de la clase `SubsetSum`, pasando el nombre del archivo a generar, el tamaño n del subset y dos enteros entre los cuales se generan los valores del subset.

Para resolver el problema, se puede utilizar el método `resolver_problema` de la misma clase, pasando el nombre del archivo generado, un t y un parametro ϵ de aproximación.

4. Código

creador_grafos.py

```
from random import randint
from grafo import Grafo


# Crea un grafo de n vertices y 2*n aristas, junto con el archivo correspondiente
# Importante: No correr el algoritmo para n <= 4, ya que para esos n, 2*n > n*(n-1)/2
# (Aristas en grafo completo), por lo que no existe grafo posible con n vertices y 2*n aristas
def crearGrafoConexo(n, nombre):
    if 2 * n > n * (n - 1) / 2:
        print 'No es posible correr el algoritmo ya que 2*n > n*(n-1)/2 para n =', n
        return
    cantAristas = 2 * n

    arch = open(nombre, 'w')
    arch.write(str(n) + "\n")
    arch.write(str(cantAristas) + "\n")
    # Escribo la primer arista, que siempre se va a corresponder con la 0 <-> 1
    arch.write(str(0) + " " + str(1) + "\n")
    # Creo el grafo inicial, junto con la primera arista
    g = Grafo()
    g.agregar_vertice(0)
    g.agregar_vertice(1)
    g.agregar_arista_no_dirigida(0, 1)

    # Primero creo un grafo conexo de n vertices y n-1 aristas, agregando de a 1 por vez
    # y creando una arista entre el y cualquiera de los anteriores
    for i in range(2, n):
        verticeRandom = randint(0, i - 1)
        arch.write(str(i) + " " + str(verticeRandom) + "\n")
        g.agregar_vertice(i)
        g.agregar_arista_no_dirigida(i, verticeRandom)
    # Despues genero las n+1 aristas restantes de manera aleatoria, teniendo en cuenta
    # que no se creen aristas repetidas
    i = 0
    while i <= n:
        verticeRandom1 = randint(0, n - 1)
        verticeRandom2 = randint(0, n - 1)
        if verticeRandom1 != verticeRandom2 and not g.son_vecinos(verticeRandom1,
             verticeRandom2):
            arch.write(str(verticeRandom1) + " " + str(verticeRandom2) + "\n")
            g.agregar_arista_no_dirigida(verticeRandom1, verticeRandom2)
            i += 1
    arch.close()
    return g

# Crea un grafo completo con n vertices, junto con el archivo correspondiente
def crearGrafoCompleto(n, nombre):
    g = Grafo()
    for i in range(n):
        g.agregar_vertice(i)
    cantAristas = n * (n - 1) / 2
    arch = open(nombre, 'w')
```

```
    arch.write(str(n) + "\n")
    arch.write(str(cantAristas) + "\n")
    for i in range(n):
        for j in range(i + 1, n):
            g.agregar_arista_no_dirigida(i, j)
            arch.write(str(i) + " " + str(j)+ "\n") # Crea una arista del vertice i al vertice j
    arch.close()
    return g
```

grafo.py

```
PRIMERO = 0
SEGUNDO = 1
TERCERO = 2


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)


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"""
        return self.aristas

    def devolver_aristas_list(self):
        lista = []
        for i in self.aristas:
            for j in self.aristas[i].values():
                lista += j
        return lista

    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"""
        arista = Arista(id1, id2, peso)
        if id2 in self.aristas[id1]:
            self.aristas[id1][id2].append(arista)
        else:
            self.aristas[id1][id2] = [arista]

    def son_vecinos(self, id1, id2):
        """Devuelve si id1 y id2 son vecinos"""
        try:
            if self.aristas[id1][id2]:
                return True
            return False
        except:
            return False

    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 borrar_vertice(self, id):
        self.vertices.remove(id)
        del self.aristas[id]

    def borrar_arista_no_dirigida(self, id1, id2):
        self.borrar_arista_dirigida(id1, id2)
        self.borrar_arista_dirigida(id2, id1)

    def borrar_arista_dirigida(self, id1, id2):
        if self.son_vecinos(id1, id2):
            del self.aristas[id1][id2]

    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(" ")
                peso = 0
                if len(numeros) > 2:
                    peso = numeros[TERCERO].rstrip('\n')
                else:
                    numeros[SEGUNDO] = numeros[SEGUNDO].rstrip('\n')
                if dirigido:
                    self.agregar_arista_dirigida(int(numeros[PRIMERO]), int(numeros[SEGUNDO
                        ]), peso)
                else:
                    self.agregar_arista_no_dirigida(int(numeros[PRIMERO]), int(numeros[
                        SEGUNDO]), peso)
            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)
```

karger.py

```
from creador_grafos import crearGrafoConexo, crearGrafoCompleto
from parser import Parser
from os.path import isfile
from random import randint
from math import log, sqrt, pow, ceil
from copy import deepcopy
from itertools import combinations
import sys
import time

INFINITO = float("inf")

"""Correr como python karger.py n, con n la cantidad de vertices del grafo"""


class Karger(object):
    def __init__(self, g, m):
        # Como llamo 2 veces, necesito que el grafo original g se mantenga intacto para el segundo
            llamado
        self.grafo = deepcopy(g)
        self.m = m
        # Aca se guardara el set al que pertenezcan
        self.corte = {i: [i] for i in self.grafo.devolver_vertices()}

    def contraer(self, id1, id2):
        for destino, aristas in self.grafo.devolver_aristas()[id2].items():
            # Me fijo que cantidad de aristas hay entre ellos
            cantidad_aristas = len(aristas)
```

```
            # Borro las aristas originales del grafo
            self.grafo.borrar_arista_no_dirigida(id2, destino)
            # Si la arista a agregar no es una arista entre id1 e id2, agrego tantas como saque
            if destino != id1:
                for _ in range(cantidad_aristas):
                    self.grafo.agregar_arista_no_dirigida(id1, destino)
        # Borro el vertice que contraje, que se fusiono con id1
        self.grafo.borrar_vertice(id2)
        # Actualizo el set al que pertenece el vertice id2
        self.corte[id1] += self.corte[id2]
        # Borro la key id2 del diccionario
        self.corte.pop(id2, None)

    def karger(self):
        if self.grafo.devolver_cant_vertices() == self.m:
            return self.grafo
        aristas = self.grafo.devolver_aristas_list()
        aristaRandom = aristas[randint(0, len(aristas) - 1)]
        # De esta forma me aseguro que id1 siempre sea menor que id2
        self.contraer(min(aristaRandom.id1, aristaRandom.id2), max(aristaRandom.id1,
            aristaRandom.id2))
        return self.karger()


def grafoInicial(n):
    path = '../out/grafoKarger.txt'
    # Verifico que el archivo, si ya esta creado, sea de la misma cantidad de nodos
    if isfile(path) and int(open(path).readline()) == n:
        p = Parser()
        return p.leer_grafo_no_dirigido(path)
    else:
        return crearGrafoConexo(n, path)
        # return crearGrafoCompleto(n, path)


def tamanoCorte(particion1, particion2, aristas):
    tamano = 0
    for i in particion1:
        for j in particion2:
            if i in aristas and j in aristas[i]:
                tamano += len(aristas[i][j])
    return tamano


def fuerzaBruta(g):
    vertices = set(g.devolver_vertices())
    aristas = g.devolver_aristas()
    # Genero todos los subsets de vertices
    particiones = [set(j) for i in range(len(vertices)) for j in combinations(vertices, i + 1)]
    particiones.remove(vertices)  # Saco el subset que es exactamente vertices
    tamanoMin = INFINITO
    for particion in particiones:
        corte = tamanoCorte(particion, vertices - particion, aristas)
        if corte < tamanoMin:
            tamanoMin = corte
    return tamanoMin
```

```
def mejorEstimacion(g):
    n = g.devolver_cant_vertices()
    if n > 8:
        k1 = Karger(g, int(ceil(n / sqrt(2) + 1)))
        X1 = mejorEstimacion(k1.karger())
        k2 = Karger(g, int(ceil(n / sqrt(2) + 1)))
        X2 = mejorEstimacion(k2.karger())
        return min(X1, X2)
    else:
        return fuerzaBruta(g)


if len(sys.argv) != 2:
    print 'Se debe especificar el parametro n como argumento. Correr python karger.py 10, por 
        ejemplo.'
    exit()
n = int(sys.argv[1])  # Cantidad de vertices
aristas_minimas = 2 * n + 1
start = time.time()
for _ in range(int(pow(log(n), 2))):
    g = grafoInicial(n)
    cant_aristas = mejorEstimacion(g)
    if cant_aristas < aristas_minimas:
        aristas_minimas = cant_aristas
end = time.time()
print str(end - start)
print 'Siendo alfa > 2:'
print 'El corte minimo encontrado es', str(aristas_minimas), 'con probabilidad al menos (1 - (1 
    /',\
    str(n), '^ 1/alfa ) ) =', str(1 - (1 / sqrt(n)))
```

parser.py

```
from grafo import Grafo

CERO = 0
UNO = 1


class Parser(object):
    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
```

pg.py

```
def compraVenta(a):
    if len(a) < 2:
        return []
    posMin = 0
    posMax = 1
    dif = a[posMax] - a[posMin]
    posMinActual = posMin
    for i in range(1, len(a)):
        if a[i] >= a[posMinActual]:
            if (a[i] - a[posMinActual]) > dif:
                dif = a[i] - a[posMinActual]
                posMin = posMinActual
                posMax = i
        else:
            posMinActual = i
    return [posMin, posMax]
```

pg_test.py

```
import random
from pg import compraVenta


def minMax(a):
    if len(a) == 2:
        return a
    min = a[0]
    max = a[1]
    for i in range(2, len(a)):
        if max < a[i]:
            max = a[i]
    before = minMax(a[1:])
    if (before[1] - before[0]) > (max - min):
        return before
    return [min, max]


def generadorArray(n):
    a = []
    for _ in range(n):
        a.append(random.randint(1, 1000))
    return a


bien = 0
n = 1000
inicial = 2
for i in range(inicial, n):
```

```
    a = generadorArray(i)
    rta = compraVenta(a)
    rtaCuadrada = minMax(a)
    if (a[rta [1]] - a[rta [0]]) == (rtaCuadrada[1] - rtaCuadrada[0]):
        bien += 1
    else:
        print a
        print rtaCuadrada
        print rta
print bien == (n - inicial)
```

subset_sum.py

```
import random
import pickle


class SubsetSum(object):
    def levantar_problema(self, nombre_archivo):
        listaNumeros = []
        with open(nombre_archivo, 'rb') as f:
            listaNumeros = pickle.load(f)
        return listaNumeros

    def generar_problema_aleatorio(self, nombre_archivo, n, numMin, numMax):
        listaNumeros = []
        if (numMin < 0) or (numMax < 0) or (numMax < numMin):
            print "Error en los enteros minimos y maximos"
            return
        for x in range(0, n):
            listaNumeros.append(random.randint(numMin, numMax))
        listaNumeros = sorted(listaNumeros)
        with open(nombre_archivo, 'wb') as f:
            pickle.dump(listaNumeros, f)

    def recortar_lista(self, listaNumeros, d):
        if not listaNumeros:
            return
        n = len(listaNumeros)
        last = listaNumeros[0]
        nuevaListaNumeros = [listaNumeros[0]]
        for i in range(1, n):
            if listaNumeros[i] > (last * (1 + d)):
                nuevaListaNumeros.append(listaNumeros[i])
                last = listaNumeros[i]
        return nuevaListaNumeros

    def resolver_problema(self, nombre_archivo, t, e):
        listaNumeros = self.levantar_problema(nombre_archivo)
        if not listaNumeros:
            return
        n = len(listaNumeros)
        L = range(0, n + 1)
        L[0] = [0]
        for i in range(1, n):
            nuevaLista = map(lambda x: x + listaNumeros[i - 1], L[i - 1])
```

```
        L[i] = list(set(L[i - 1] + nuevaLista))
        L[i] = self.recortar_lista(L[i], e / (2 * n))
        L[i] = [j for j in L[i] if j <= t]
    L[n] = L[n - 1]
    z = L[-1][-1]
    return z
```