



5

Instituto Politécnico Nacional  
Escuela Superior de Computo

Análisis de algoritmos

Sandra Díaz Santiago

---

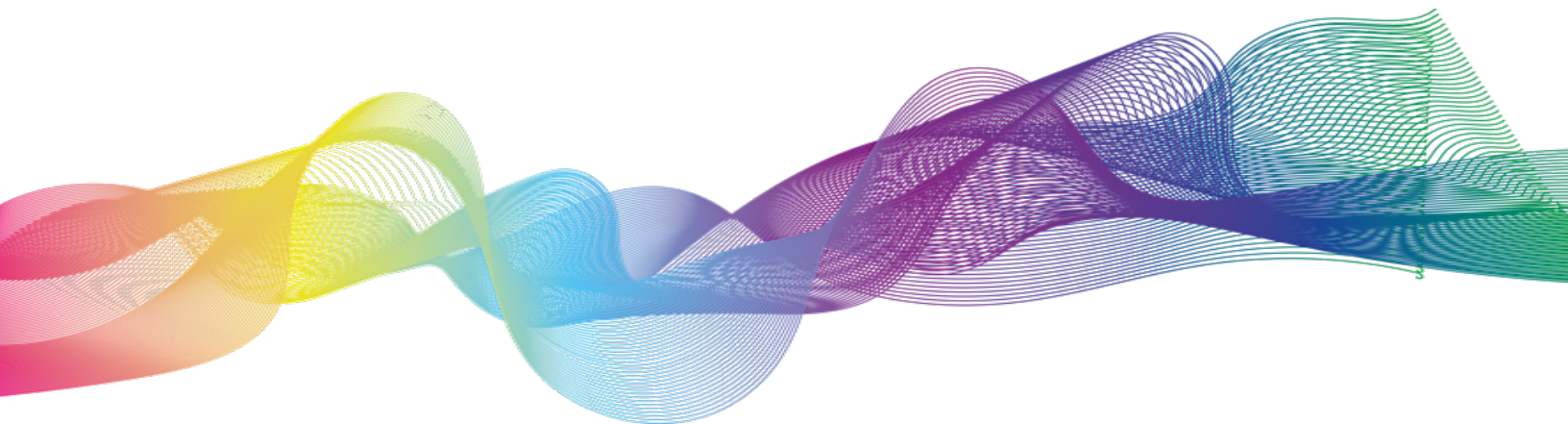
### Practica 5: Algoritmos ávidos II

---

Alumno:

- Vallejo serrano Ehecatzin.
- García González Aarón Antonio

Noviembre 04, 2019



Índice

Ejercicio 1..... 3

    Implementación..... 3

    Ejemplo ..... 6

    Ejecución..... 7

    Justificación..... 7

Ejercicio 2..... 8

    Implementación..... 8

    Ejemplo ..... 9

    Ejecución..... 10

    Justificación..... 10

## Ejercicio 1

1. Dado un grafo dirigido en el cual cada vértice  $v \in V$ , tiene un precio  $P_v$  asociado (un entero positivo). Define el costo como sigue: para cada vértice  $v \in V$ ,

***costo[v] = precio del vértice más barato alcanzable desde v incluido v mismo.***

Por ejemplo, en el grafo que se muestra a continuación (con precios en los vértices), los valores para el costo de los vértices A, B, C, D, E, F son 2,1,4,1,4,5, respectivamente.

Diseña un algoritmo eficiente que llene el arreglo completo con el costo para cada vértice. Considera que el grafo debe ser a cíclico. Implementa tu algoritmo en el lenguaje de programación de tu elección.

## Implementación

Se implementa en el lenguaje de programación Python, para la manipulación de grafos se empleo el modulo networkx.

A continuación del código ejecutable (main):

```
# ----- Importación de módulos a utilizar
from utilidades_grafos import *
from archivo import *

# ----- Declaración de apuntador a archivo
name_file_INPUT = open("input.txt", "r")
name_file_OUTPUT = open("output.txt", "w")

# ----- Ejecutable
G = nx.DiGraph()
lista_de_nodos_valor = leer_vertices_valor(name_file_INPUT)
agregarVerticeValor(lista_de_nodos_valor,G)
lista_de_aristas = leer_aristas(name_file_INPUT)
G.add_edges_from(lista_de_aristas)
costos_recorrido = obtenerCostoPorVertice(G)
imprimirEnArchivo(name_file_OUTPUT,costos_recorrido)
print("Operacion exitosa, archivo generado: [output.txt]\n")

name_file_INPUT.close()
name_file_OUTPUT.close()
```

Funcionamiento:

1. Donde lo primero que se realiza es abrir los archivos de origen de los datos del grafo así como el archivo a crear donde serán almacenados los resultados del programa.
2. Creamos el grafo dirigido (dígrafo)
3. Leemos los vértices desde el archivo origen, y añadimos al grafo previamente creado únicamente los identificadores de los vértices (es decir, sin su ponderación o peso)

```
lista_de_nodos_valor = leer_vertices_valor(name_file_INPUT)
G.add_nodes_from(list(nodo[0] for nodo in lista_de_nodos_valor))
```

4. Leemos las aristas desde el archivo origen, y añadimos al grafo previamente creado la lista de tuplas con dichas aristas [(vertice\_origen\_id, vertice\_destino\_id), ...]

```
lista_de_aristas = leer_aristas(name_file_INPUT)
G.add_edges_from(lista_de_aristas)
```

5. Finalmente calculamos los valores por cada nodo y los mandamos a el archivo definido al inicio.

```
costos_recorrido = obtenerCostoPorVertice(G, lista_de_nodos_valor)
imprimirEnArchivo(name_file_OUTPUT, costos_recorrido)
```

Nota: Ver módulos:

```
from utilidades_grafos import *
from archivo import *
```

A continuación el modulo utilidades\_grafos

```
import networkx as nx
```

```
# Función que lista todos los vértices que son iniciales o no accesibles desde otro vértice
# Recibe dos argumentos (1: lista de aristas (a,b) y 2: lista de nodos)
# Retorna una lista con los identificadores de los vértices que cumplen
```

```
def generarNodosIniciales(grafo):
    # Requerimos saber los vértices de inicio, ya que no es cíclico
    # Los nodos de inicio tienen salidas pero no entradas
    lista_aristas = grafo.edges()
    lista_nodos = list(nodo[0] for nodo in grafo.nodes())
    nodos_id = {}
    llaves = []
    resultado = []

    for x in list(nodo[0] for nodo in lista_nodos):
        nodos_id[x] = 0

    llaves = list(nodos_id.keys())
    for arista in lista_aristas:
        if(arista[0] in llaves):
            nodos_id[arista[0]] += 1
        if(arista[1] in llaves):
            nodos_id[arista[1]] += 1

    return list( k for k, v in nodos_id.items() if v == 1)
```

```
# Función que agrega vértices a un grafo, vértice, peso
# recibe como argumento la lista de vértices y el mismo grafo
# no regresa nada
```

```
def agregarVerticeValor(lista, grafo):
    for vertice in lista:
        grafo.add_node(vertice[0], weight = vertice[1])
```

```

# Función que obtiene el menor precio dado un nodo final
# recibe 3 argumentos, 1: el nodo destino, 2: el mismo grafo y 3: lista de los nodos con valor
# retorna una tupla con el camino a seguir y el valor del mismo
def obtenerMenorPrecioParticular(nodo_destino,grafo):
    nodos_iniciales = generarNodosIniciales(grafo)
    dic_valor_nodos = dict(grafo.nodes(data='weight', default=1))
    conexiones = nx.single_target_shortest_path(grafo,nodo_destino)
    candidatos = {}
    candidatos_valor = {}
    contador = None

    for clave in conexiones:
        if clave in nodos_iniciales:
            candidatos[clave] = conexiones[clave]

    for clave in candidatos:
        contador = 0
        for camino in candidatos[clave]:
            contador += dic_valor_nodos[camino]
        candidatos_valor[contador] = candidatos[clave]

    return min(candidatos_valor.keys()),candidatos_valor[min(candidatos_valor.keys())]

# Función que obtiene el menor precio de cada nodo
# recibe 2 argumentos, 1: el mismo grafo y 2: lista de los nodos con valor
# retorna una tupla con el camino a seguir y el valor del mismo
def obtenerCostoPorVertice(grafo):
    resultados = []
    vertices = grafo.nodes()
    for vertice in vertices:
        resultados.append(list(vertice)+list(obtenerMenorPrecioParticular(vertice,grafo)))

    return resultados

```

Y finalmente el modulo de archivo

```

# Función que lee la primera línea de un archivo
# Recibe como argumento el archivo
# Retorna la lista de tuplas ([ID_nodo,valor],...)
def leer_vertices_valor(archivo):
    linea = archivo.readline()
    lista_linea = linea.split()
    nuevo = []
    for dato in lista_linea:
        cadena = dato.replace(',', '')
        nuevo.append((cadena[0],int(cadena[1])))

```

```

return nuevo

# Función que le de la línea 1 a la n
# recibe como argumento el archivo
# retorna lista de tuplas con las aristas
# Esta función se manda a llamar después de leer_vertices_valor
def leer_aristas(archivo):
    lineas = archivo.readlines()

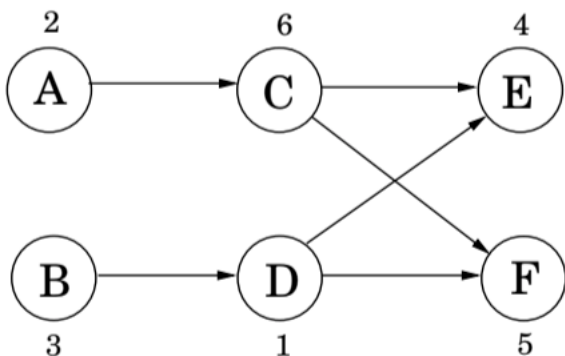
    nuevo = []
    for linea in lineas:
        lista_linea = linea.split()
        for dato in lista_linea:
            nuevo.append(tuple(dato.replace(',', '')))
    return nuevo

# Función que recibe como parámetros el archivo donde se va a imprimir
# y la lista de precios por cada vértice con formato para su visualización
# No hay valor de retorno
def imprimirEnArchivo(archivo_destino, lista_resultado):
    archivo_destino.write("Arreglo completo con el costo para cada vértice\n")
    archivo_destino.write("costo[vertex] = precio mas barato alcanzable desde un vertice generador\n")
    incluido el mismo\n")

    archivo_destino.write("ID vertice\tCosto minimo\t\tCamino a recorrer\n")
    for vertice in lista_resultado:
        archivo_destino.write('{0:2s}{3} \t\t\t{3}{1:3d}{3} \t\t\t\t{3}{2:10s}{3}'.format(vertice[0], vertice[1], str(vertice[2]), ''))
        archivo_destino.write("\n")

```

## Ejemplo



Vértice origen	Vértice destino	Valor de recorrido	Camino recorrido
A	A	2	A
A	B	-	-
A	C	8	A, C
A	D	-	-
A	E	12	A, C, E
A	F	13	A, C, F
B	A	-	-
B	B	3	B
B	C	-	-
B	D	4	B, D
B	E	8	B, D, E
B	F	9	B, D, F

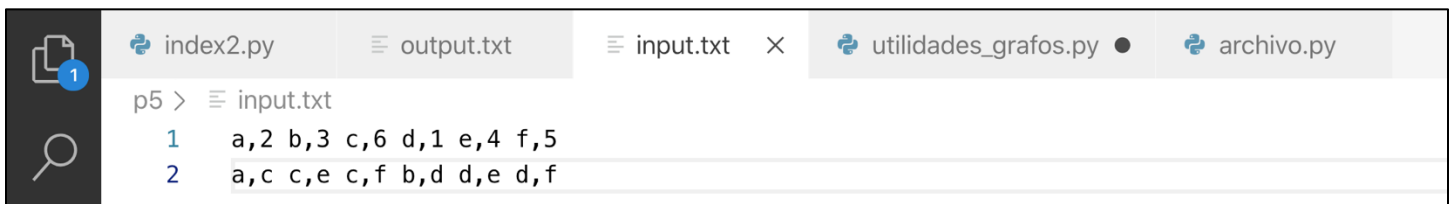
Dado que se nos solicita llegar a todos los nodos destino, claramente en la imagen podemos observar que hay nodos que son inaccesibles si no son estos mismos los que pudiesen ser el inicio del grafo, el grafo no se podría recorrer.

A la izquierda la tabla con todos los posibles caminos que hay para cada vértice.

## Ejecución

```
[MacBook-Pro-de-Aaron:p5 aarongarcia$ python3 index2.py  
Operacion exitosa, archivo generado: [output.txt]
```

## Miramos el archivo origen



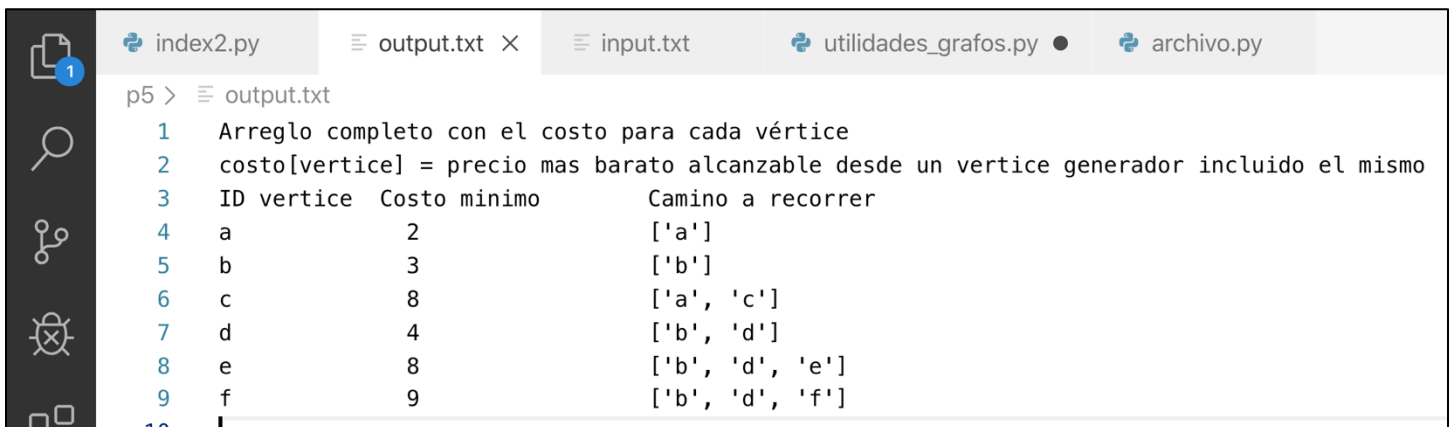
```
p5 > input.txt  
1 a,2 b,3 c,6 d,1 e,4 f,5  
2 a,c c,e c,f b,d d,e d,f
```

Donde:

La primera línea representa los vértices con su respectivo peso

La segunda línea representa las aristas origen, destino

## Miramos el archivo destino



```
p5 > output.txt  
1 Arreglo completo con el costo para cada vértice  
2 costo[vertice] = precio mas barato alcanzable desde un vertice generador incluido el mismo  
3 ID vertice Costo minimo Camino a recorrer  
4 a 2 ['a']  
5 b 3 ['b']  
6 c 8 ['a', 'c']  
7 d 4 ['b', 'd']  
8 e 8 ['b', 'd', 'e']  
9 f 9 ['b', 'd', 'f']
```

Complejidad  $O(n^2)$

## Justificación

Dado que la función empelada `nx.single_target_shortest_path(grafo,nodo_destino)`, retorna el camino mas barato dado un nodo destino e internamente con la función con base a pesos de sus vértices y `obtenerMenorPrecioParticular(nodo_destino,grafo)` obtiene ávidamente el mas barato, podría decir que la solución es optima, además de que se cumple el objetivo de la misma.

## Ejercicio 2

2. Diseña un algoritmo ávido para encontrar el árbol recubridor máximo de un grafo, es decir el árbol recubridor con costo máximo. Justifica por qué tu algoritmo da el árbol recubridor de costo máximo. Implementa el algoritmo en el lenguaje de programación de tu elección.

### Implementación

```
from collections import defaultdict

class Grafo:
    def __init__(self, vertices):
        self.vert= vertices #Numero de vertices del grafo
        self.grafo = [] #EL grafo xd

    def addArista(self, origen, destino, peso): #se van agregando los pares de nodos al grafo
        self.grafo.append([origen, destino, peso])

    def buscar(self, padre, i):
        if padre[i] == i:
            return i
        return self.buscar(padre, padre[i])

    def union(self, padre, rango, x, y): #Esta funcion hace union de dos conjuntos de x,y
        a = self.buscar(padre, x)
        b = self.buscar(padre, y)
        if rango[a] < rango[b]: #agrega los arboles de rango menor a la raiz del de rango mayor
            padre[a] = b
        elif rango[a] > rango[b]:
            padre[b] = a
        else : #si su rango es el mismo, entonces uno se vuelve raiz y se incrementa el rango
            padre[b] = a
            rango[a] += 1

    def ARM(self):
        resultado =[]
        i = 0
        e = 0
        padre = []
        rango = []
        self.grafo = sorted(self.grafo, key=lambda item: item[2], reverse = True) #Lo del key es una
funcion lambda que hace el sort basandose en el peso (item[2])
        # donde self.grafo[i] es el de peso mayor
        for nodo in range(self.vert):
            padre.append(nodo)
            rango.append(0)
```



```

        while e < self.vert -
1 : #hace las iteraciones para ir armando el arbol (esto es el algoritmo)
        origen,destino,peso = self.grafo[i]
        i = i + 1
        x = self.buscar(padre, origen)
        y = self.buscar(padre ,destino)
        if x != y: #verificar que no se hagan ciclos
            e = e + 1
            resultado.append([origen,destino,peso])
            self.union(padre, rango, x, y)
            #print(rango)
            print(padre)

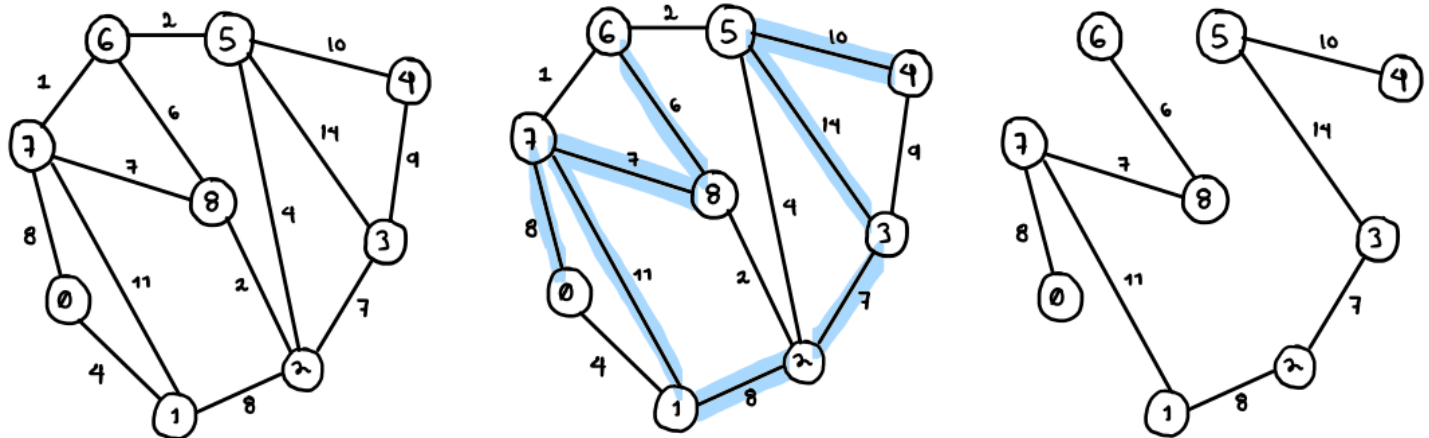
    print ("El resultado para el arbol recubridor Maximo usando Kurskal es: ")

    for origen,destino,peso in resultado:
        print ("%d -- %d == %d" % (origen,destino,peso))
    print(rango)
    print(padre)

#PRUEBAS grafo recibe el numero de vertices y se agregan las aristas en sentido origen, destino, pe
so
g = Grafo(9)
g.addArista(0, 1, 4)
g.addArista(1, 2, 8)
g.addArista(2, 3, 7)
g.addArista(3, 4, 9)
g.addArista(4, 5, 10)
g.addArista(5, 6, 2)
g.addArista(6, 7, 1)
g.addArista(7, 8, 7)
g.addArista(0, 7, 8)
g.addArista(1, 7, 11)
g.addArista(2, 8, 2)
g.addArista(6, 8, 6)
g.addArista(2, 5, 4)
g.addArista(3, 5, 14)
g.ARM()

```

## Ejemplo



## Ejecución

```
[MacBook-Pro-de-Aaron:desktop aarongarcia$ python3 kruskal.py
El resultado para el arbol recubridor Maximo usando Kurskal es:
3 -- 5 == 14
1 -- 7 == 11
4 -- 5 == 10
1 -- 2 == 8
0 -- 7 == 8
2 -- 3 == 7
7 -- 8 == 7
6 -- 8 == 6
```

## Justificación

Complejidad de tiempo:  $O((\text{numero de aristas}) \log(\text{numero de aristas}))$  u  $O((\text{numero de aristas}) \log(\text{numero de vértices}))$ . La clasificación de las aristas lleva tiempo  $O((\text{numero de aristas}) \log(\text{numero de aristas}))$ . Después de ordenar, iteramos a través de todas las aristas y aplicamos el algoritmo find-union. Las operaciones de búsqueda y unión pueden tomar casi un tiempo  $O(\log(\text{numero de vértices}))$ . Entonces, la complejidad general es el tiempo  $O((\text{numero de aristas}) \log(\text{numero de aristas}) + E \log(\text{numero de vértices}))$ . El valor de  $E$  puede ser al menos  $O(2(\text{numero de vértices}))$ , por lo que  $O(\log(\text{numero de vértices}))$  son  $O(\log(\text{numero de aristas}))$  iguales. Por lo tanto, la complejidad del tiempo general es  $O((\text{numero de aristas}) \log(\text{numero de aristas}))$  u  $O((\text{numero de aristas}) \log(\text{numero de vértices}))$ .