UNIVERSIDAD NACIONAL DE COLOMBIA
SEDE BOGOTÁ
FACULTAD DE INGENIERÍA

Algoritmos
Germán J. Hernández

Taller 3
Grafos, Complejidad Computacional, Programación Dinámica

Miguel Angel Ortiz Marin - 1032504680

1. Para los 3 grafos, calcule (por inspección) la longitud los caminos mı́as cortos desde el nodo 0 hacia el resto de nodos.

Para el grafo a:

nodo	1	2	3	4
distancia a 0	5	3	6	8

Para el grafo b:

nodo	1	2	3	4
distancia a 0	5	3	6	8

Para el grafo c:

nodo	1	2	3	4
distancia a 0	2	8	5	7

2. Para el grafo b, ¿puede encontrar un camino desde 0 a 1 de costo 0? Explique.

Sí.
Si se sigue la secuencia: 0 -> 2 -> 1 -> 3 -> 2 -> 1 varias veces la longitud del camino se reduce en 1 por cada vuelta que se hace por el ciclo negativo 2->1->3->2. como la longitud del camino sin hacer ciclos es 5 solo basta con realizar el ciclo 5 veces.

3.

Código:

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jun  4 16:17:38 2018

@author: Miguel Angel
"""


class edge:

    def __init__(self, u, v, w):
        self.u = u
        self.v = v
        self.w = w


edgesa = [edge(0,1,9),
          edge(0,2,3),
          edge(2,1,2),
          edge(1,3,1),
          edge(2,3,6),
          edge(3,2,4),
          edge(2,4,6),
          edge(3,4,2)]


edgesb = [edge(0,1,9),
          edge(0,2,3),
          edge(2,1,2),
          edge(1,3,1),
          edge(2,3,6),
          edge(3,2,-4),
          edge(2,4,6),
          edge(3,4,2)]

edgesc = [edge(0,1,2),
          edge(0,2,8),
          edge(2,1,2),
          edge(1,3,3),
          edge(2,3,6),
          edge(3,2,4),
          edge(2,4,6),
          edge(3,4,2)]
```

```
def BellmanFord(vertices, edges, source, print_=False):
    distance = [0]*vertices
    predecessor = [0]*vertices

    '''
    This implementation takes in a graph, represented as
    lists of vertices and edges, and fills two arrays
    (distance and predecessor) with shortest-path
    (less cost/distance/metric) information
    '''
    # Step 1: initialize graph
    for v in range(vertices):
        distance[v] = 10000000
        predecessor[v] = -1

    distance[source] = 0

    for i in range(vertices - 1):
        for edge in edges:
            u = edge.u
            v = edge.v
            w = edge.w
            if distance[u] + w < distance[v]:
                distance[v] = distance[u] + w
                predecessor[v] = u
        print(distance)

    for edge in edges:
        u = edge.u
        v = edge.v
        w = edge.w
        if distance[u] + w < distance[v]:
            return "Graph contains a negative-weight cycle", distance,
predecessor

    return distance, predecessor


print(BellmanFord(5, edgesa, 0))

print(BellmanFord(5, edgesb, 0))

print(BellmanFord(5, edgesc, 0))
```

Output al correrlo (python 3):
[0, 5, 3, 6, 8]
[0, 5, 3, 6, 8]
[0, 5, 3, 6, 8]
[0, 5, 3, 6, 8]
([0, 5, 3, 6, 8], [-1, 2, 0, 1, 3])

[0, 5, 2, 6, 8]
[0, 4, 1, 5, 7]
[0, 3, 0, 4, 6]
[0, 2, -1, 3, 5]
('Graph contains a negative-weight cycle', [0, 2, -1, 3, 5], [-1, 2, 3, 1, 2])

[0, 2, 8, 5, 7]
[0, 2, 8, 5, 7]
[0, 2, 8, 5, 7]
[0, 2, 8, 5, 7]
([0, 2, 8, 5, 7], [-1, 0, 0, 1, 3])

4. El grafo b contiene un ciclo negativo, lo cual el algoritmo Bellman-Ford detecta y manda un error.
5. En los tres grafos se le hicieron (vértices-1)*edges = 4*8 = 32 llamados a la sección de código que corresponde a la función relax. Debido a la construcción del algoritmo no se pueden realizar menos llamados sin comprometer la correctitud del algoritmo. Sin embargo se podría aplicar métodos voraces como el algoritmo de Dijkstra para realizar los llamados en un orden más eficiente y además cuando en alguna iteración no hayan ocurrido ningún cambio en las distancias significa que el algoritmo ha encontrado la respuesta.
6. A manera de dijkstra:

Para el grafo a:
0->2
2->1
1->3
3->4
el resto de arcos que se recorran no importa el orden ya que este ya habrá encontrado la respuesta óptima para cada nodo.

Para el grafo c:
0->1
1->3
3->2
2->1
3->4

Los dos últimos en cualquier orden