Escuela de Ingeniería Eléctrica

Estructuras de Datos Abstractas y Algoritmos para Ingeniería

Proyecto de investigación
Algoritmo de Johnson

Edward Cerdas Rodríguez - B71956

14 de noviembre del 2020

Índice

1. Introducción **3**

2. Discusión **3**

2.1. ¿Cómo funciona el Algoritmo de Johnson? 3

2.2. Para qué se utiliza 3

2.3. Cuál es su comportamiento en términos de complejidad. 4

2.4. Implementación del algoritmo en Python 4

2.5. Corridas de ejemplo con diferentes valores 6

2.5.1. Ejemplo 1 6

2.5.2. Ejemplo 2 7

2.5.3. Ejemplo 3 9

3. Conclusiones **10**

1. Introducción

Para conocer el origen de los grafos, nos remontamos al año 1736, en donde estos nacerían gracias a Leonhard Euler, quien intentaba resolver el famoso problema de los puentes de Konisberg. Dos grandes aportes para que la teoría de grafos sea tal y como la conocemos hoy en día son la de Gustav Kirchoff en 1845, con sus investigaciones acerca de los circuitos de calculo de voltaje y el aporte de Francis Guthrie en 1852, con la famosa hipótesis de los cuatro colores.

El Algoritmo de Johnson debe su nombre al investigador científico de computación Donald B. Johnson, de nacionalidad estadounidense, quien obtuvo su doctorado en 1973 bajo la supervisión de David Gries en la Universidad Privada de Investigación de Cornell (Cornell University) en New York. Johnson es conocido también por crear una estructura de datos llamada d-heap, las cuales son colas de datos con prioridad. (1)

El Algoritmo de Johnson es una manera de encontrar el camino más corto posible entre todos los pares de vértices de un grafo dirigido disperso. Permite que las aristas tengan pesos negativos, pero no permite ciclos de pesos negativos.

Para lograr esto, el algoritmo de Johnson utiliza el algoritmo de Bellman-Ford, transformando el grafo inicial, eliminando todas las aristas de peso negativo. Cuando quita estos pesos negativos utiliza el algoritmo de Dijsktra en este nuevo grafo. (2)

2. Discusión

2.1. ¿Cómo funciona el Algoritmo de Johnson?

Los pasos para ejecutar el algoritmo son los siguientes. (2)

1. Se añade un nuevo nodo al grafo existente y le asignamos un nombre arbitrario, que para este ejemplo será "q", este nuevo nodo se conecta con todos los nodos ya existentes por medio de aristas de peso 0.

2. Se utiliza el algoritmo de Bellman-Ford tomando como punto de inicio al nodo "qz como destino a todos los demás nodos, de uno en uno. De esta forma se determinará, para cada nodo, el peso mínimo de este camino que une al nodo "qçon el nodo destino.

3. A las aristas del grafo original se les cambia el peso por un nuevo valor que puede ser calculado con la información obtenida en el paso 2 y la siguiente formula:

$$h(u) - h(v) + w(u, v) \tag{1}$$

   Donde h(u) representa el peso generado por el algoritmo de Bellman-Ford para el nodo inicial, h(v) el peso generado por el algoritmo de Bellman-Ford para el nodo destino y w(u,v) el peso original de la arista.

4. Una vez cambiados los pesos de las aristas, eliminando los pesos negativos, para cada uno de los nodos se puede utilizar el algoritmo de Dijkstra para determinar el camino más corto entre ese nodo original y los otros nodos en este nuevo grafo.

2.2. Para qué se utiliza

Este algoritmo puede ser utilizado para resolver problemas de talleres de flujo, en los que el trabajo debe ser dividido entre dos o más máquinas de la forma más eficiente posible. El principal objetivo de este algoritmo es el minimizar el tiempo requerido para el procesamiento de la totalidad del trabajo (3).

2.3. Cuál es su comportamiento en términos de complejidad.

Si bien es cierto, las computadoras de hoy en día son capaces de realizar millones de cálculos por segundo, no todos los calculos y resoluciones de problemas son tan sencillos, algunos de ellos pueden tardar incluso años en resolverse. Es aquí donde entra en juego la complejidad computacional, la cual mide la cantidad de cálculos según los datos de entrada del algoritmo, ya que medir el tiempo exacto que le tomará al algoritmo realizar todo el proceso es muy difícil para problemas muy complejos. (4)

Dado que para completar el algoritmo de Johnson se deben utilizar otros dos algoritmos, de forma que el algoritmo de Bellman-Ford se ejecuta una vez y el algoritmo de Dijkstra se ejecuta una cantidad V de veces durante el proceso. Entonces se tiene que para el algoritmo de Bellman-Ford la complejidad temporal es de O(VE), mientras que para Dijkstra es de O(VLogV). Entonces, juntando los dos algoritmos, la complejidad temporal para el Algoritmo de Johnson es de Entonces, O(V2logV + VE).

2.4. Implementación del algoritmo en Python

```
1
2  # Algoritmo de Johnson
3
4  from collections import defaultdict
5  MAX_INT = float('Inf')
6
7  #Retorna el vertice con la minima distancia desde el origen
8  def minDistancia(dist, visitado):
9
10         (minimo, minnodo) = (MAX_INT, 0)
11         for nodo in range(len(dist)):
12                 if minimo > dist[nodo] and visitado[nodo] == False:
13                         (minimo, minnodo) = (dist[nodo], nodo)
14
15         return minnodo
16
17
18 #Se aplica el algoritmo de Dijkstra para el grafo con los pesos negativos removidos
19 def Dijkstra(grafo, grafoModificado, src):
20
21         # Numero de vertices en el grafo
22         num_vertices = len(grafo)
23
24
25         sptSet = defaultdict(lambda : False)
26
27         # Distancia mas corta de todos los nodos desde el origen
28         dist = [MAX_INT] * num_vertices
29
30         dist[src] = 0
31
32         for count in range(num_vertices):
33                 nodoActual = minDistancia(dist, sptSet)
34                 sptSet[nodoActual] = True
35
36                 for nodo in range(num_vertices):
37                         if ((sptSet[nodo] == False) and
38                                 (dist[nodo] > (dist[nodoActual] +
39                                 grafoModificado[nodoActual][nodo])) and
40                                 (grafo[nodoActual][nodo] != 0)):
41
```

```
                                dist[nodo] = (dist[nodoActual] + grafoModificado[
                                    nodoActual][nodo]);

        # Imprime la distancia mas corta desde el origen
        for nodo in range(num_vertices):
                print ('Nodo ' + str(nodo) + ': ' + str(dist[nodo]))

# Funcion para cualcular la distancia mas corta desde el origen hasta todos los otros
    vertices usando el algoritmo de Bellman-Ford
def BellmanFord(aristas, grafo, num_vertices):

        # Se agrega el nodo s como nodo origen y se calculan las distancias minimas
            hacia los otros nodos
        dist = [MAX_INT] * (num_vertices + 1)
        dist[num_vertices] = 0

        for i in range(num_vertices):
                aristas.append([num_vertices, i, 0])

        for i in range(num_vertices):
                for (src, des, peso) in aristas:
                        if((dist[src] != MAX_INT) and
                                        (dist[src] + peso < dist[des])):
                                dist[des] = dist[src] + peso

        return dist[0:num_vertices]

# Funcion para implementar el Algoritmo de Johnson
def Johnson(grafo):

        aristas = []

        # Se crea una lista de aristas para el algoritmo de Bellman-Ford
        for i in range(len(grafo)):
                for j in range(len(grafo[i])):

                        if grafo[i][j] != 0:
                                aristas.append([i, j, grafo[i][j]])

        # Se modifican los pesos
        modificarPesos = BellmanFord(aristas, grafo, len(grafo))

        grafoModificado = [[0 for x in range(len(grafo))] for y in
                                        range(len(grafo))]

        # Se eliminan los pesos negativos
        for i in range(len(grafo)):
                for j in range(len(grafo[i])):

                        if grafo[i][j] != 0:
                                grafoModificado[i][j] = (grafo[i][j] +
                                                modificarPesos[i] - modificarPesos[j])
                                                    ;

        print ('Grafo Modificado por Bellman-Ford: ' + str(grafoModificado))

        # Se ejecuta el algoritmo de Dijkstra para cada nodo uno a uno
        for src in range(len(grafo)):
                print ('\nDistancia mas corta entre el nodo ' +
```

```
                                                    str(src) + ' y el:\n')
                Dijkstra(grafo, grafoModificado, src)

# Se crea el grafo para trabajar con el
grafo =          [[0,-3, 2, 3],
                  [0, 0, 4, 0],
                  [0, 0, 0, 1],
                  [0, 0, 0, 0]]

Johnson(grafo) # Se aplica el metodo a el grafo que se creo anteriormente
```

2.5. Corridas de ejemplo con diferentes valores

2.5.1. Ejemplo 1

```
Grafo Modificado por Bellman-Ford: [[0, 0, 2, 3], [0, 0, 1, 0], [0, 0, 0, 1], [0, 0, 0, 0]]

Distancia mas corta entre el nodo 0 y cada uno de los demás nodos:

Nodo 0: 0
Nodo 1: 0
Nodo 2: 1
Nodo 3: 2

Distancia mas corta entre el nodo 1 y cada uno de los demás nodos:

Nodo 0: inf
Nodo 1: 0
Nodo 2: 1
Nodo 3: 2

Distancia mas corta entre el nodo 2 y cada uno de los demás nodos:

Nodo 0: inf
Nodo 1: inf
Nodo 2: 0
Nodo 3: 1

Distancia mas corta entre el nodo 3 y cada uno de los demás nodos:

Nodo 0: inf
Nodo 1: inf
Nodo 2: inf
Nodo 3: 0
>
```

Figura 1: Corrida de ejemplo #1

En la figura 1 se muestran los resultados del programa utilizando como entrada el grafo representado con la siguiente matriz de adyacencia:

[0,-3, 2, 3]
[0, 0, 4, 0]
[0, 0, 0, 1]
[0, 0, 0, 0]

Se utilizaron valores sencillos para esta primer prueba, en la cual se puede ver que el programa trabaja tal y como se solicita, retornando los valores del nuevo grafo modificado por el algoritmo de Bellman-Ford y luego las mínimas distancias que provee el algoritmo de Dijkstra.

Los resultados marcados en amarillo son para destacar que algunos de estos datos tienen el valor de infinito debido a que no existe un camino que conecte los nodos en la dirección deseada, tal es el caso de cualquier nodo intentando conectarse con el nodo 0, ya que este grafo en específico luce tal y como se muestra en la figura 2.

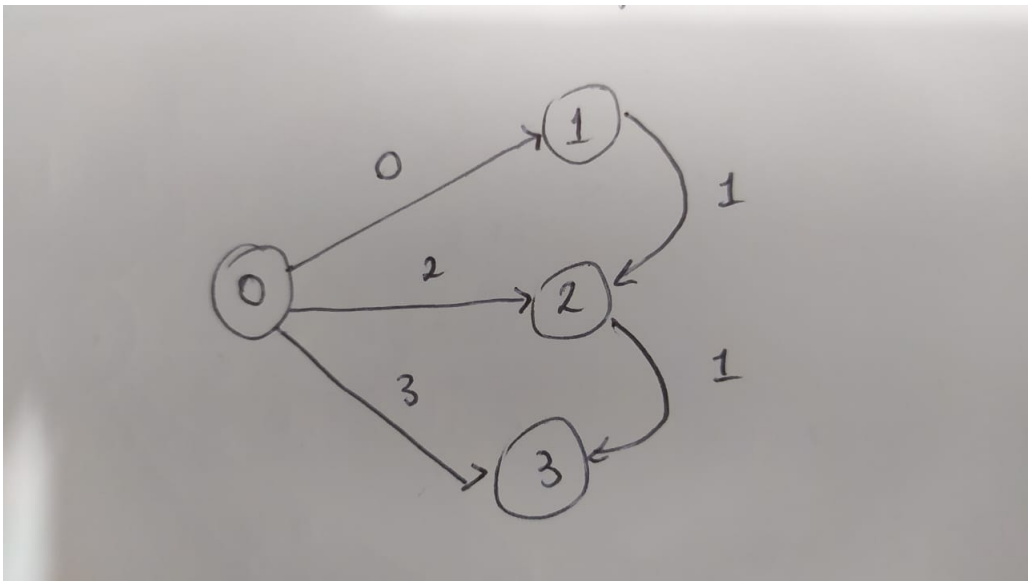

Figura 2: Grafo para ejemplo #1

Como se puede apreciar, no existe un camino que se dirija hacia el nodo 0, el único valor que se obtiene para un nodo intentando conectar con este nodo 0, es el valor del propio nodo 0 conectandose consigo mismo, de esta manera el camino se marca con un 0, ya que no tiene que moverse a ninguna parte.

2.5.2. Ejemplo 2

Para continuar con un ejemplo diferente, se puso a prueba el programa con otros valores, dando múltiples caminos para relacionar los nodos, lo que se espera es que las distancias marcadas como infinito desaparezcan, debido a que todos los nodos tendrán al menos un camino para llegar a todos los demás nodos.

Se utilizaron los siguientes valores en la matriz de adyacencia para esta corrida:

[0,-3, 2, 3]
[-6, 0, 4,10]
[0, 5,0, 1]
[-1, 2, 0, 0]

Con estos valores el grafo se ve tal y como se muestra en la figura 3.

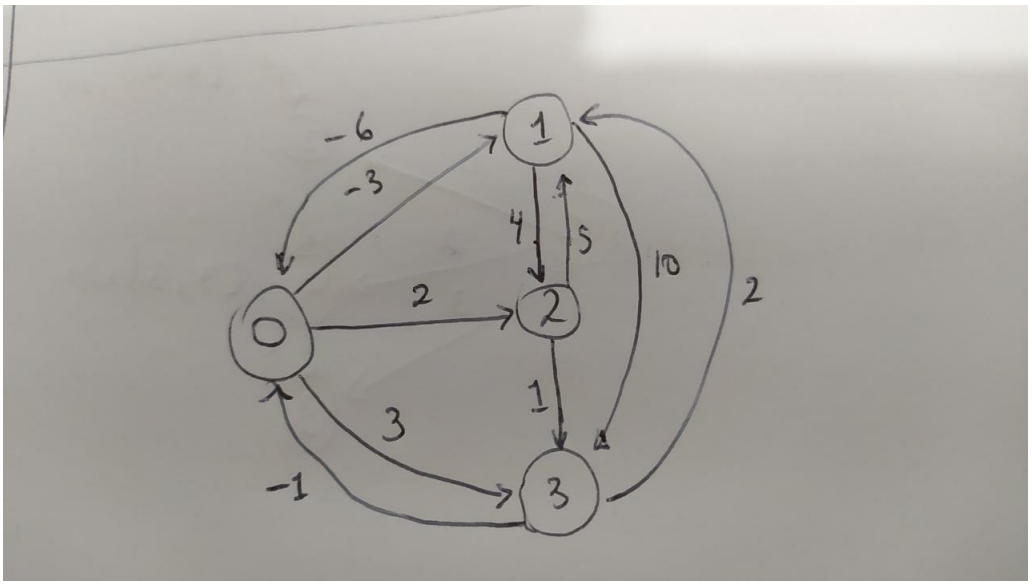

Figura 3: Grafo para ejemplo #2

Con estos datos de entrada, se obtuvieron los datos de la figura 4

```
Grafo Modificado por Bellman-Ford: [[0, -9, -8, -8], [0, 0, 0, 5], [0, 9, 0, 0], [10, 7, 0, 0]]

Distancia mas corta entre el nodo 0 y cada uno de los demás nodos:

Nodo 0: 0
Nodo 1: -9
Nodo 2: -9
Nodo 3: -9

Distancia mas corta entre el nodo 1 y cada uno de los demás nodos:

Nodo 0: 0
Nodo 1: 0
Nodo 2: -8
Nodo 3: -8

Distancia mas corta entre el nodo 2 y cada uno de los demás nodos:

Nodo 0: 7
Nodo 1: 7
Nodo 2: 0
Nodo 3: 0

Distancia mas corta entre el nodo 3 y cada uno de los demás nodos:

Nodo 0: 7
Nodo 1: 7
Nodo 2: -1
Nodo 3: 0
>
```

Figura 4: Corrida de ejemplo #2

Como se puede observar, lo esperado se cumplió, ya no existen estos caminos de extensión infinita, ya que todos

los nodos pueden acceder a los demás nodos sin problemas.

2.5.3. Ejemplo 3

En este tercer ejemplo lo que se busca es aislar uno de los nodos, de forma que no tenga conexión con ningún otro nodo, esperando que todos los caminos se marquen como infinito. La matriz de adyacencia ingresada es la siguiente:

[0, 0, 0, 0]
[0, 0, 4,10]
[0, 5, 0, 1]
[0, 2, 0, 0]

Esta matriz de adyacencia es resultado de un grafo tal y como el que se muestra en la figura 5,

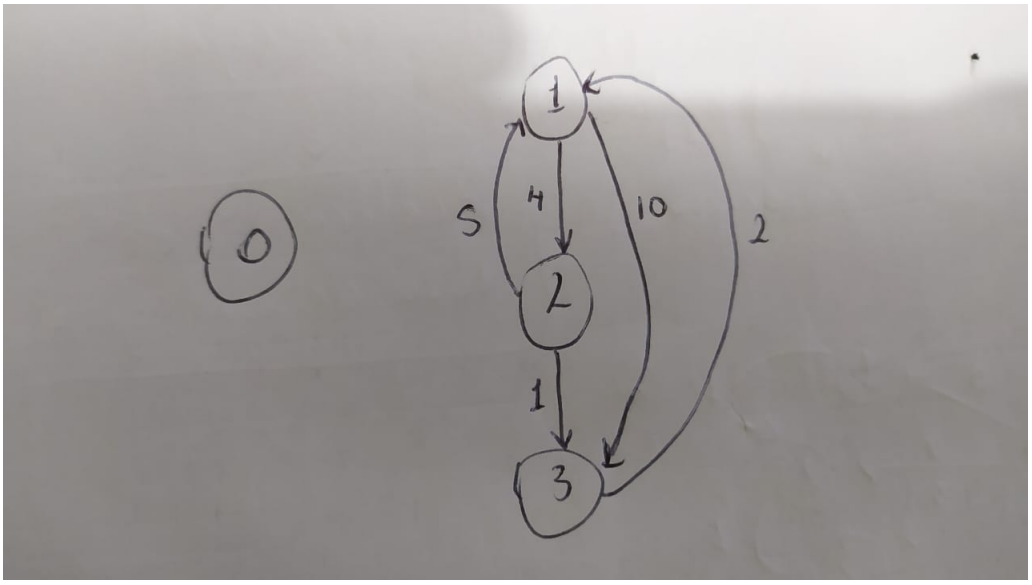

Figura 5: Grafo para ejemplo #3

En este grafo se puede apreciar que el nodo 0 está aislado, por lo que se espera que todos los caminos relacionados a este nodo tengan extensión infinita. En la figura 6 se muestran los resultados para esta entrada de datos.

```
Grafo Modificado por Bellman-Ford: [[0, 0, 0, 0], [0, 0, 4, 10], [0, 5, 0, 1], [0, 2, 0, 0]]

Distancia mas corta entre el nodo 0 y cada uno de los demás nodos:

Nodo 0: 0
Nodo 1: inf
Nodo 2: inf
Nodo 3: inf

Distancia mas corta entre el nodo 1 y cada uno de los demás nodos:

Nodo 0: inf
Nodo 1: 0
Nodo 2: 4
Nodo 3: 5

Distancia mas corta entre el nodo 2 y cada uno de los demás nodos:
Nodo 0: inf
Nodo 1: 3
Nodo 2: 0
Nodo 3: 1

Distancia mas corta entre el nodo 3 y cada uno de los demás nodos:

Nodo 0: inf
Nodo 1: 2
Nodo 2: 6
Nodo 3: 0
>
```

Figura 6: Corrida de ejemplo #3

Como se puede observar en la figura 6, las predicciones fueron correctas, todos los caminos que se dirigen o provienen del nodo 0 no existen, por tanto su extensión es infinita. El único camino existente es el que empieza en el nodo 0 y termina en el nodo 0, el cual tiene una extensión de 0, como es de esperar.

3. Conclusiones

El Algoritmo de Johnson es muy útil a la hora de resolver problemas relacionados con la eficiencia en talleres de flujo, donde los procesos son secuenciales. El algoritmo ayuda a encontrar la manera más óptima de repartir la carga de trabajo entre las máquinas.

Al utilizar herramientas de programación, para crear programas en lenguajes como Python, se puede ejecutar el algoritmo siempre y cuando se tenga conocimiento sobre las formas de representar a los grafos como matrices.

La matriz de adyacencia es una de las formas más útiles y sencillas de representar a los grafos como matrices.

Referencias

[1] D. B. Johnson, "Priority queues with update and finding minimum spanning trees", Information Processing Letters, 4: 53–57, 1975.

[2] H. Allaoui, A. Artiba, "Johnson's algorithm: a key to solve optimally or approximately flowshop scheduling problems with unavailability periods". International Journal of Production Economics, 2009.

[3] C. Rapine Erratum to scheduling of a two-machine flowshop with availability constraints on the first machine. International Journal of Production Economics, http://dx.doi. org/10.1016/j.ijpe.2012.11.006

[4] N. Ziviani, "Diseño de Algoritmos con implementaciones en Pascal y C"Madrid: Thomson Editores España, 2007.