

Programación III

TRABAJO PRÁCTICO OBLIGATORIO

GRUPO de TRABAJO  
“La barba de Godio”

TEMAS  
Algoritmo de Floyd  
Problema del Viajante

PROFESOR: Lic. Esteban Calabria

|  |  |
| --- | --- |
| Integrante | Legajo |
| CARIATI, Yamila Soledad | 1064010 |
| MALDONADO, Martín | 1056424 |
| RICCOMBENI, Maximiliano | 1017172 |
| VIEIRO, Javier Martín | 1062141 |

Segundo Cuatrimestre 2015

Viernes Noche

# Índice

[1. Índice 1](#_Toc434163084)

[2. Enunciado 2](#_Toc434163085)

[2.1. Algoritmo de Floyd 2](#_Toc434163086)

[2.2. Problema del Viajante 2](#_Toc434163087)

[3. Resolución 3](#_Toc434163088)

[3.1. Algoritmo de Floyd 3](#_Toc434163089)

[3.2. Problema del Viajante 4](#_Toc434163090)

# Enunciado

## Algoritmo de Floyd

Se desea que el alumno investigue y documente el algoritmo de Floyd. Se desea que desarrolle los siguientes puntos. Cada uno de los siguientes puntos debe figurar como un apartado distinto de este documento.

* Explicación con sus propias palabras del algoritmo citando fuentes de donde obtuvo la información. Para poder aprobar este punto el grupo debe demostrar poder realizar una investigación exhaustiva y a conciencia y de calidad profesional.
* Backtracking
  + Pseudocódigo de backtracking que aplique la lógica del algoritmo de floyd
  + Realizar un TDA grafo adecuado para la implementación
  + Análisis de complejidad del punto anterior
  + Implementación en java
* Programación dinámica
  + Pseudocódigo
  + Análisis de complejidad del mismo
  + Implementación en java

## Problema del Viajante

Se desea investigar y entender el famoso problema del viajante y su relación con la complejidad algorítmica. Se desea completar los siguientes puntos:

* Definición del problema
* Implicancias del problema a nivel complejidad temporal
* Pseudocódigo de una resolución del problema
* Análisis de complejidad del pseudocódigo anterior
* Implementación en java del pseudocódigo propuesto

# Resolución

## Algoritmo de Floyd

# Creado por Bernard Roy en 1959 es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados.

# El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución. El algoritmo de Floyd-Warshall es un ejemplo de programación dinámica, teniendo en cuenta que este tipo de programación tiene como fin encontrar una solución óptima a dicho problema recursivamente.

# Compara todos los posibles caminos a través del [grafo](https://es.wikipedia.org/wiki/Grafo) entre cada par de vértices. El algoritmo es capaz de hacer esto con sólo *V3* comparaciones. Lo hace mejorando paulatinamente una estimación del camino más corto entre dos vértices, hasta que se sabe que la estimación es óptima

# Características:

# Obtiene la mejor ruta (menor distancia) entre todo par de nodos.

# Trabaja con la matriz D inicializada con las distancias directas entre todo par de nodos.

# Matriz D:

# 8d4a7da4332bfb32f8f8d6b48e7c2100.png

# Donde

# Valor de Arista si existe arista entre Vi y Vj

D[i][j] ∞ si no existe arista entre Vi y Vj

0 si i = j

# La iteración se produce sobre nodos intermedios, o sea para todo elemento de la matriz se prueba si lo mejor para ir de i a j es a través de un nodo intermedio elegido o como estaba anteriormente, y esto se prueba con todos los nodos de la red. Una vez probados todos los nodos de la red como nodos intermedios, la matriz resultante da la mejor distancia entre todo par de nodos.

# Hasta no hallar la última matriz no se encuentran las distancias mínimas.

# Su complejidad es del orden de V3.

# En otras palabras, dado un grafo G dirigido y ponderado, para calcular el camino mínimo entre dos vértices cualesquiera del grafo, se puede aplicar el algoritmo de Floyd que, dada la matriz L de adyacencia del grafo G, calcula una matriz D con la longitud del camino mínimo que une cada par de vértices.

# Como se dijo con anterioridad, este algoritmo puede ser considerado de Programación Dinámica ya que es aplicable el principio de óptimo, que puede enunciarse para este problema de la siguiente forma: si en el camino mínimo de vi a vj , vk es un vértice intermedio, los caminos de vi a vk y de vk a vj han de ser a su vez caminos mínimos. Por lo tanto, puede plantearse la relación de recurrencia que resuelve el problema como:

# *Dk(i, j) = Mink−1{Dk−1(i, k), Dk−1(k, j)}*

**Fuente: http://www.ecured.cu/index.php/Floyd-Warshall**

### Backtracking

#### Pseudocódigo

**Algoritmo**

Floyd

**Entrada**

Vértice Orig de origen

Vértice Dest de destino

Cantidad de escalas Esc, originalmente el número de vértices del grafo

**Salida**

**Pseudocódigo**

Res <-- ?

#### TDA grafo

public interface GrafoDirTDA<E> {

public void InicializarGrafo();

public ConjuntoTDA<E> Vertices();

public void AgregarVertice(E var1);

public void AgregarArista(E var1, E var2, int var3);

public E Elegir();

public boolean ExisteArista(E var1, E var2);

public int PesoArista(E var1, E var2);

public void EliminarVertice(E var1);

public void EliminarArista(E var1, E var2);

public ConjuntoTDA<E> Adyacentes(E var1);

}

#### Complejidad

#### Implementación Java

### Programación dinámica

#### Pseudocódigo

**Algoritmo**

Floyd

**Entrada**

G grafo de entrada

**Salida**

R grafo con una arista con la distancia mínima entre cada par de vértices

**Pseudocódigo**

inicializarGrafo(R) **C1**

copiarGrafo(G, R) **C2**

para cada k en G.Vertices **V1**

para cada i de G.Vertices **V2**

para cada j de G.Vertices **V3**

si i!=j Y existeArista(R, i, k) Y existeArista(R, k, j) **C3**

si existeArista(R, i, j) **C4**

si pesoArista(R, i, k) + pesoArista(R, k, j) < pesoArista(R, i, j) **C5**

agregarArista(R, i, j, pesoArista(R, i, k) + pesoArista(R, k, j)) **C6**

fin si

sino

agregarArista(R, i, j, pesoArista(R, i, k) + pesoArista(R, k, j)) **C7**

fin si

fin si

fin para

fin para

fin para

devolver R **C8**

#### Complejidad

O = C1-2-8 + V1 \* ( V2 \* ( V3 \* ( C3-4 + Max(C5+C6, C7) ) )

O = C + V3 \* C

O = V3 Cúbico

#### Implementación Java

public static GrafoDirTDA <Integer> floyd(GrafoDirTDA <Integer> g) {

ConjuntoTDA <Integer> conjuntoI, conjuntoJ, conjuntoK;

int i, j, k;

GrafoDirTDA <Integer> r = new GrafoDir < Integer > ();

r.InicializarGrafo();

// Copio el grafo original

conjuntoK = g.Vertices();

while (!conjuntoK.conjuntoVacio()) {

k = conjuntoK.elegir();

conjuntoK.sacar(k);

r.AgregarVertice(k);

}

conjuntoK = g.vertices();

while (!conjuntoK.conjuntoVacio()) {

k = conjuntoK.elegir();

conjuntoK.sacar(k);

ConjuntoI = g.Adyacentes(k);

While(!conjuntoI.conjuntoVacio()) {

i = conjuntoI.elegir();

conjuntoI.sacar(i);

r.AgregarArista(k, i, g.PesoArista(k, i));

}

}

conjuntoK = g.Vertices();

while (!conjuntoK.conjuntoVacio()) {

k = conjuntoK.elegir();

conjuntoK.sacar(k);

conjuntoI = g.Vertices();

conjuntoI.sacar(k);

while (!conjuntoI.conjuntoVacio()) {

i = conjuntoI.elegir();

conjuntoI.sacar(i);

if (r.ExisteArista(i, k)) {

conjuntoJ = r.Adyacentes(k);

conjuntoJ.sacar(i);

while (!conjuntoJ.conjuntoVacio()) {

j = conjuntoJ.elegir();

conjuntoJ.sacar(j);

if (r.ExisteArista(i, j)) {

if (r.PesoArista(i, k) + r.PesoArista(k, j) < r.Pesoarista(i, j)) {

r.AgregarArista(i, j, r.PesoArista(i, k) + r.PesoArista(k, j);

}

} else {

r.AgregarArista(i, j, r.PesoArista(i, k) + r.pesoarista(k, j));

}

}

}

}

}

return r;

}

## Problema del Viajante

|  |
| --- |
| Travelling Salesman Problem |
| *Fuente: xkcd (http://www.xkcd.com/399/)* |

El problema del viajante es uno de los problemas más famosos de la matemática computacional. Pertenece a una serie de problemas que parecen tener una fácil solución pero en la práctica presentan una gran dificultad. Este problema en concreto ha sido muy estudiado por sus múltiples aplicaciones en la optimización de recursos, tanto en el campo empresarial (logística de transporte) como en el de la robótica (desplazamientos que se realizan al hacer un circuito impreso).

Problema del viajante

Plantea cómo un viajante podría empezar y terminar en una ciudad concreta, pasando por todas las ciudades que están en su mapa una sola vez, y por la mínima ruta posible. A priori la solución puede parecer sencilla, solo habría que probar cuál de las posibles combinaciones de rutas sería la óptima, lo que llamamos por “fuerza bruta”. La dificultad aparece cuando el número de ciudades es elevado, las posibles combinaciones aumentan de manera exponencial.

Los problemas cuyo aumento de datos hacen que el tiempo de resolución (o computación) aumente exponencialmente, se les llama **NP-completos**. Por tanto, el problema del viajante es NP-completo, ya que un aumento de ciudades eleva exponencialmente el número de combinaciones posibles y, debido a esto, el número de pruebas que hay que realizar para ver cuál es la combinación óptima, incrementando exponencialmente el tiempo de resolución. Aquí estriba su impedimento en la práctica, si el número de ciudades es elevado no existe computadora a nuestro alcance capaz de computar (o solucionar) el problema en un tiempo razonable.

Pongamos un ejemplo, imaginemos un problema del viajante con 20 ciudades, las posibles combinaciones de estas ciudades serían más de 2,4 trillones (20!). Supongamos que tenemos el mejor microprocesador que hay en el mercado, lo que nos posibilita un mayor número de instrucciones por segundo. En este caso vamos a imaginar que tenemos un Intel Quad Core (cuatro núcleos) con unas 49 millones de instrucciones por segundo (49.000 MIPS). En el mejor de los casos, y siendo conscientes que esto no es así en la realidad, consideremos que necesitamos una sola instrucción por cada combinación y nuestro procesador trabaja exclusivamente en el problema del viajante. Con una sencilla cuenta de división (combinaciones divididas entre MIPS) podemos darnos cuenta que se resolvería en más de 1.500 años.

(ese último párrafo no me convence)

**Fuente: http://queaprendemoshoy.com/problema-del-viajante/**

Algunos links que estuve viendo (filtré bastante):

<http://www.dc.uba.ar/materias/aed3/2013/1c/teorica/algo3_metah_heu.pdf>

<http://stackoverflow.com/questions/22977748/complexity-for-greedy-algo-travelling-salesman-and-nearest-neighbor-search>

<http://www-eio.upc.es/~nasini/Blog/TSP_Notes.pdf>

En este comenta las diferentes formas de encarar el problema:

<http://www.academia.edu/3828405/IMPLEMENTATION_OF_HEURISTICS_FOR_SOLVING_TRAVELLING_SALESMAN_PROBLEM_USING_NEAREST_NEIGHBOURANDNEARESTINSERTIONAPPROACHES>

y la guía de apuntes de la cátedra

Me parece que deberíamos elegir el nearest neighbor, pero la decisión se las dejo a ustedes.