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

1. Índice

2. Enunciado

2.1. 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

2.2. 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

3. Resolución

3.1. Algoritmo de Floyd

3.1.1. Descripción del problema

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 entre cada par de vértices. El algoritmo es capaz de hacer esto con sólo V^3 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:

$$D = \begin{bmatrix} 0 & 3 & 5 & 1 & \infty & \infty \\ 3 & 0 & \infty & \infty & 9 & \infty \\ 5 & \infty & 0 & 7 & 7 & 1 \\ 1 & \infty & 7 & 0 & \infty & 4 \\ \infty & 9 & 7 & \infty & 0 & \infty \\ \infty & \infty & 1 & 4 & \infty & 0 \end{bmatrix}$$

Donde

D[i][j] si i = j

Valor de Arista	si existe arista entre Vi y Vj
∞	si no existe arista entre Vi y Vj

- 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 V^3.

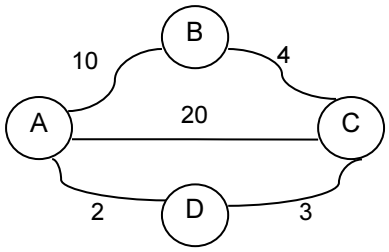
En otras palabras, dado un grafo G dirigido y ponderado, para calcular el camino mínimo entre dos vértices cualesquiera del grafo, se puede aplicar el algoritmo de Floyd que, dada la matriz L de adyacencia del grafo G, calcula una matriz D con la longitud del camino mínimo que une cada par de vértices.

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 v_i a v_j , v_k es un vértice intermedio, los caminos de v_i a v_k y de v_k a v_j han de ser a su vez caminos mínimos. Por lo tanto, puede plantearse la relación de recurrencia que resuelve el problema como:

$D_k(i, j) = Min_{k-1}\{D_{k-1}(i, k), D_{k-1}(k, j)\}$

3.1.2. Ejemplo Práctico

Dado un Grafo:



Armar la Matriz de Adyacencia E_0

E_0	A	B	C	D
A	0	10	20	2
B	10	0	4	∞
C	20	4	0	3
D	2	∞	3	0

Verificar si conviene ir directamente de un vértice a otro o hacer una escala en otro vértice en el medio.

$DE_1(A,B) = Min(DE_0(A,B), DE_0(A,C) + DE_0(C,B), DE_0(A,D) + DE_0(D,B) = Min(10, 20+4, 2+\infty) = 10$
$DE_1(A,C) = Min(DE_0(A,C), DE_0(A,B) + DE_0(B,C), DE_0(A,D) + DE_0(D,C) = Min(20,10+4, 2+3) = 5$
$DE_1(A,D) = Min(DE_0(A,D), DE_0(A,C) + DE_0(C,D), DE_0(A,B) + DE_0(B,D) = Min(2, 20+3, 10+\infty) = 2$
$DE_1(B,C) = Min(DE_0(B,C), DE_0(B,A) + DE_0(A,C), DE_0(B,D) + DE_0(D,C) = Min(4, 10+20, \infty+3) = 4$
$DE_1(B,D) = Min(DE_0(B,D), DE_0(B,A) + DE_0(A,D), DE_0(B,C) + DE_0(C,D) = Min(\infty, 10+2, 4+3) = 7$

$DE_1(C,D) = Min(DE_0(C,D), DE_0(C,A) + DE_0(A,D), DE_0(C,B) + DE_0(B,D) = Min(3, 20+2, 4+\infty) = 3$

E_1	A	B	C	D
A	0	10	**5**	2
B	10	0	4	**7**
C	**5**	4	0	3
D	2	**7**	3	0

En la Matriz E_1 se almacena el peso del camino más corto entre un par de vértices con a lo sumo una escala en el medio.

Se continúa con este proceso repitiendo las mismas operaciones con la Matriz E_x para obtener la Matriz E_{x+1}.

$DE_2(A,B) = Min(10, 5+4, 2+7) = 9$
$DE_2(A,C) = Min(5,10+4, 2+3) = 5$
$DE_2(A,D) = Min(2, 5+3, 10+7) = 2$
$DE_2(B,C) = Min(4, 10+5, 7+3) = 4$
$DE_2(B,D) = Min(7, 10+2, 4+3) = 7$
$DE_2(C,D) = Min(3, 5+2, 4+7) = 3$

E2	A	B	C	D
A	0	**9**	5	2
B	**9**	0	4	7
C	5	4	0	3
D	2	7	3	0

Finalmente se obtiene una Matriz con todos los valores del camino mínimo entre cualquier par de vértices. Se diferencia del Algoritmo de Dijkstra ya que éste solo calcula una sola fila (sólo un vértice a otro) de la Matriz calculada mediante el Algoritmo de Floyd.

3.1.3. Backtracking

3.1.3.1. Pseudocódigo

Algoritmo
FloydBT
Entrada
Grafo G(V,A)
Vértice O de origen
Vértice D de destino
Cantidad E de escalas G.V
Algoritmo estaConectado(Origen, Destino, Grafo)
Algoritmo Distancia(Origen, Destino, Grafo)
Salida
Valor mínimo entre O y D
Pseudocódigo

```
Res <- infinito
Si (E = 0)
     Si (estaConectado(O, D, G))
          Res <- Distancia(O, D, G)                                    Constante
Sino
     Res <- FloydBT(O, D, G, E-1, estaConectado)                       T(V-1)
     Para cada V en G.V
          Si(V <> O y V <> D)
               Res <- Min(Res, FloydBT(G, O, V, E-1, estaConectado) + FloydBT(G, V, D, E-1,
               estaConectado))
          Fin si
```

```
      Fin para
Fin sino
Devolver Res
```

3.1.3.2. TDA GrafoDir

```
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);
}
```

3.1.3.3. Complejidad

$$T(V) \begin{cases} C & \text{si } E = 0 \\ T(V-1) + C & \text{si } E > 0 \end{cases}$$

$T(V) = T(V-1) + V = T(V-2) + 2V = T(V-3) + 3V = T(V-K) + K.V$

Caso base: $T(V-K) = T(0) \Rightarrow T = T(V-V) + V * V = V^2$
$V - K = 0$

$V = K$

La complejidad es de V^2.

3.1.3.4. Implementación Java

```
import TDA.*;

public final class FloydBacktracking {

	public static int INF = 99999;

	public static int FloydBT(GrafoDirTDA<Integer> grafo, int origen, int destino, int escala) {
		int res = INF;
		if (escala == 0){
			if (grafo.ExisteArista(origen, destino))
				res = grafo.PesoArista(origen, destino);
		}
		else
		{
			res = FloydBT(grafo, origen, destino, escala-1);
			ConjuntoTDA<Integer> vert = grafo.Vertices();
			int medio;
			while (!vert.conjuntoVacio()) {
				medio = vert.elegir();
				if(medio != origen && medio != destino){
					res = Math.min(res, FloydBT(grafo, origen, medio, escala - 1) + FloydBT(grafo, medio, destino,
					escala - 1));
				}
				vert.sacar(medio);
			}
		}
		return res;
```

```
    }
}
```

3.1.4. Programación dinámica

3.1.4.1. Pseudocódigo

Algoritmo

Floyd

Entrada

Matriz A de adyacencia del grafo G
Matriz D con los valores de los caminos mínimos (inicialmente vacía)
Grafo G(V,A)

Salida

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

Pseudocódigo

```
V <- #G.V                                                    C1
Para i de 1…V                                                V1
     Para j de 1…V                                           V2
          D[i,j] = A[i,j]                                    C2
     Fin para
Fin para
Para k de 1…V                                                V3
     Para i de 1…V                                           V4
          Para j de 1…V                                      V5
               D[i,j] = Min(D[i,j], D[i,k] + D[k,j])         C3
          Fin para
     Fin para
```

```
Fin para
Devolver D
```

C_4

3.1.4.2. Complejidad

$O = C_{1-4} + V_1 * (V_2 * C_2) + V_3 * (V_4 * (V_5 * C_3)$

$O = C + V^2 * C + V^3 * C$

$O = V^3$ ← Cúbico

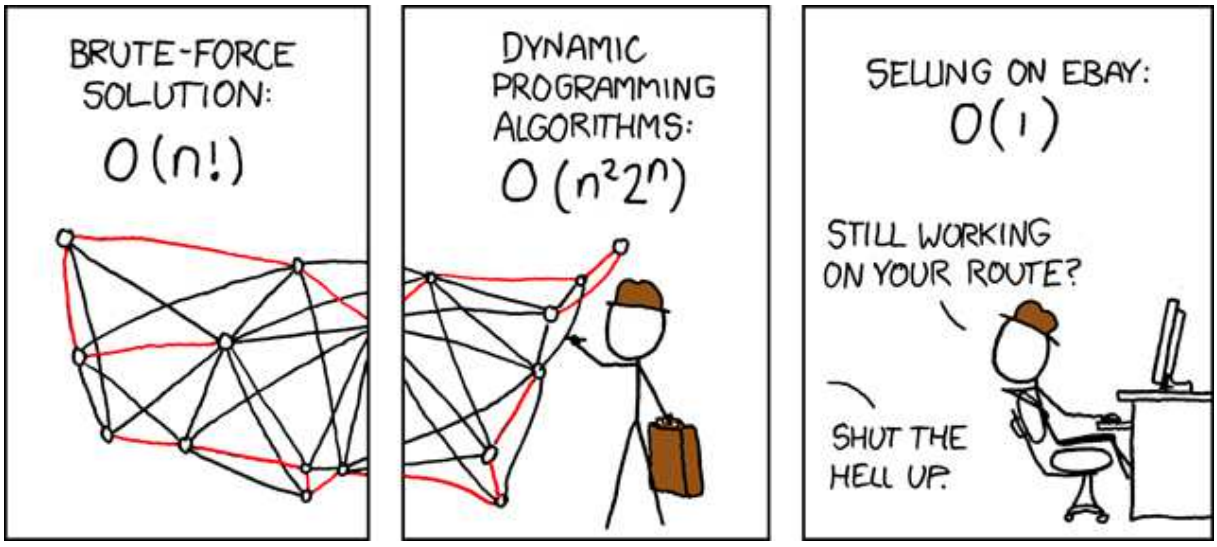
3.1.4.3. Implementación Java

```
public final class FloydDinamica {

	public static int INF = 99999;

	public static void floydMatrices(int[][] matrizAdy) {
		int V = matrizAdy[0].length;
		int dist[][] = new int[V][V];
		int i, j, k;
		// Inicializar la matriz solucion igual que la matriz del grafo.
		for (i = 0; i < V; i++) {
			for (j = 0; j < V; j++) {
				dist[i][j] = matrizAdy[i][j];
			}
		}
		// Añadir vertices uno a uno para establecer los vertices intermedios
		// Al inicio de la iteracion: {0, 1, 2, .. k-1} vertices intermedios
```

```
            // Al final de la iteracion: se agrega vertice k
            for (k = 0; k < V; k++) {
                  // Por cada vertice de origen
                  for (i = 0; i < V; i++) {
                        // Por cada vertice destino de cada vertice de origen
                        for (j = 0; j < V; j++) {
                              // Si el vertice k es el camino más corto de i a j,
                              // actualizar el valor de dist[i][j]
                              if (dist[i][k] + dist[k][j] < dist[i][j]) {
                                    dist[i][j] = dist[i][k] + dist[k][j];
                              }
                        }
                  }
            }
            imprimirSolucion(dist);
            return;
      }

      private static void imprimirSolucion(int dist[][])
      {
            int V = dist[0].length;
            System.out.println("La siguiente matriz muestra el camino más corto entre cada par de vértices:");
            for (int i = 0; i < V; ++i)
            {
                  for (int j = 0; j < V; ++j)
                  {
               if (dist[i][j] == INF)
                   System.out.print("INF ");
               else
                   System.out.print(dist[i][j]+"   ");
            }
            System.out.println();
        }
    }
}
```

3.2. Problema del Viajante

3.2.1. Descripción del problema



Fuente: xkcd (http://www.xkcd.com/399/)

El problema del viajante (en inglés, Traveling Salesman Problem o TSP) es uno de los problemas más famosos de la matemática computacional. Es de origen incierto y aparece en libros de viajantes del siglo XIX, pero no tenían ningún tratamiento matemático en ellos. Los primeros en formular el problema matemáticamente fueron W.R. Hamilton y británico Thomas Kirkman. El "Juego Icosian" de Hamilton era un rompecabezas basado en encontrar un ciclo de Hamilton (un camino dentro de un grafo que comience y termine en el mismo vértice y que pase por todos los vértices). La forma general fue estudiada por primera vez por los matemáticas en Viena y Harvard, durante los años 1930s, destacándose Karl Menger, quien definió los problemas, considerando el obvio algoritmo de "fuerza bruta".

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).

El problema describe a un viajante de comercio que debe visitar "n" ciudades. Cada ciudad está conectada con las restantes mediante carreteras de longitud conocida. Consiste en hallar la longitud de la ruta que deberá tomar para visitar todas las ciudades retornando a la ciudad de partida, pasando una única vez por cada ciudad y de modo tal que la longitud del camino recorrido sea mínima.

Existen multitud de variantes al problema de viajante general, como por ejemplo:

- **MAX-TSP:** Consiste en encontrar un circuito hamiltoniano de costo máximo.

- **TSP con cuello de botella:** Consiste en encontrar un circuito hamiltoniano tal que minimice el mayor costo de entre todas las aristas del mismo, en vez de minimizar el costo total.

- **TSP gráfico:** Consiste en encontrar un circuito de costo mínimo tal que se visiten las ciudades al menos una vez.

- **TSP agrupado:** Los nodos o ciudades están divididos en "clúster" o grupos, de manera que lo que se busca es un circuito hamiltoniano de costo mínimo en el que se visiten los nodos de cada grupo de manera consecutiva.

- **TSP generalizado:** Los nodos o ciudades también están divididos en grupos, pero lo que se busca es un circuito de costo mínimo que visite exactamente un nodo de cada grupo.

- **TSP con múltiples viajantes:** Existen un número "m" de viajantes, cada uno de los cuales debe visitar algunas de las ciudades. El problema se transforma, por tanto, en la búsqueda de una partición de los nodos a visitar X_1, ..., X_m y de "m" ciclos, uno para cada X_i , de manera que la suma de las distancias recorridas por los "m" viajantes sea mínima.

Dentro de los algoritmos utilizados para solucionar este problema, podemos clasificarlos en:

Algoritmos exactos:

- **Fuerza bruta:** consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo.

- **Held-Karp:** La idea de este método es que en un circuito óptimo, tras haber recorrido una serie de ciudades, el camino que atraviese las restantes ciudades debe ser también óptimo. Este hecho permite construir el circuito paso a paso: a partir de una lista de los caminos de mínimo costo entre las ciudades de todos los subconjuntos de tamaño k, especificando en cada caso el origen y el destino, se puede crear una lista de todos esos caminos para conjuntos de k ciudades. Held y Karp demostraron que este algoritmo era capaz de resolver problemas del viajante de cualquier tamaño n en un tiempo que era, como máximo, proporcional a $n^2 2^n$. El principal inconveniente de este método es la cantidad de tiempo computacional que requiere, restringiendo su uso a pequeños problemas, incluso con la tecnología actual.

- **Ramificación y Poda (Branch & Bound):** el problema se subdivide en dos problemas, de tal manera que el primero de ellos sea el problema anterior más una restricción adicional, y el segundo sería de nuevo el problema original más otra restricción. Ante la imposibilidad de resolver alguno de estos problemas, o ante la posibilidad de que la solución obtenida no sea válida, los subproblemas pueden ser, a su vez, divididos en dos subproblema. Estos subproblemas reciben el nombre de ramificaciones, pues generalmente este método se representa por un árbol de decisión, siendo la raíz el problema original y las ramas los subproblemas. El proceso continúa de esta forma hasta obtener alguna solución satisfactoria o hasta que alguna rama sea "podada", lo que ocurre cuando el subproblema correspondiente da lugar a una solución tal que la función objetivo tome un valor superior a algún elemento anterior del ´árbol.

Algoritmos heurísticos: algoritmos que dan lugar a soluciones casi-óptimas.

- **Vecino más próximo:** la clave de este algoritmo es siempre visitar la ciudad más cercana. Consiste en (1) seleccionar una ciudad de forma aleatoria (2) Encontrar la ciudad no visitada más cercana e ir allí (3) Si existe alguna ciudad no visitada restante, repetir el punto 2 (4) Volver a la primera ciudad.

- **Greedy:** construye gradualmente un recorrido seleccionando repetidamente la arista más corto y agregándolo al recorrido mientras no cree un ciclo con menos de N aristas, o incrementar los grados de cualquier nodo a más de 2. No se puede agregar la misma arista dos veces obviamente.

- **Inserción Heurística:** hay muchas variantes para elegir. Principalmente consiste en empezar con un recorrido de un sub-set de ciudades, y luego insertar el resto mediante una heurística.

- **Christofides:** La mayoría de los heurísticos garantizan un radio de peores casos de 2. Christofides extendió uno de estos algoritmos y concluyó que el radio de peores casos es de 3/2.

Existen optimizaciones tales como "Búsqueda Tabú", "Algoritmo Genético" para que el algoritmo mejore en aspectos de su complejidad temporal.

3.2.2. Implicancias del problema a nivel complejidad temporal

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**. Esto significa que no se ha encontrado ningún algoritmo que logre resolverlos en un tiempo polinómico. 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.

En el problema se presentan N! rutas posibles, aunque se puede simplificar ya que dada una ruta nos da igual el punto de partida y esto reduce el número de rutas a examinar en un factor N quedando (N-1)!.

Por otro lado, se puede continuar simplificando el problema sosteniendo que como no importa la dirección en que se desplace el viajante, el número de rutas a examinar se reduce nuevamente en un factor 2. Por lo tanto, hay que considerar (N-1)!/2 rutas posibles.

En la práctica, para un problema del viajante con 5 ciudades hay 12 rutas diferentes y no necesitamos un ordenador para encontrar la mejor ruta, pero apenas aumentamos el número de ciudades las posibilidades crece exponencialmente (en realidad, factorialmente):

- Para 10 ciudades hay 181.440 rutas diferentes.
- Para 30 ciudades hay más de $4 * 10^{31}$ rutas posibles. Un ordenador que calcule un millón de rutas por segundo necesitaría 10^{18} años para resolverlo. Dicho de otra forma, si se hubiera comenzado a calcular al comienzo de la creación del universo (hace unos 13.400 millones de años) todavía no se habría terminado.

Puede comprobarse que por cada ciudad nueva que incorporemos, el número de rutas se multiplica por el factor N y crece exponencialmente (factorialmente). Por ello el problema pertenece a la clase de problemas NP-completos.

La solución más directa puede ser, intentar todas las permutaciones (combinaciones ordenadas) y ver cuál de estas es la menor (usando una Búsqueda de fuerza bruta). El tiempo de ejecución es un factor polinómico de orden O(n!), el Factorial del número de ciudades, esta solución es impracticable para dado solamente 20 ciudades.

3.2.3. Pseudocódigo de una resolución del problema

Resolución por Fuerza Bruta

Algoritmo

```
EjecutarFuerzaBruta
```

Entrada

```
Clase Viajero
      - Atributo (cantCiudades)
      - Algoritmo imprimirRutaCostoMinimo
      - Algoritmo calcularCostoDeViajeEntreCiudades
Algoritmo chequearRuta
```

Salida

```
El detalle de la mejor ruta entre Ciudad 0, pasando por todas las ciudades, y regresando nuevamente a la
ciudad 0
```

Pseudocódigo

```
Ruta <- int[viajero.cantCiudades]
minRuta <- int[viajero.cantCiudades]
minCosto <- -1
contador <- 0
ruta[0] <- 0 //la primera ciudad es siempre 0
Para i=0 hasta viajero.cantCiudades
      ruta[1]<-i
      chequearRuta(ruta, 2)
Fin para
Imprimir en pantalla ("Resultado de Fuerza Bruta")
Imprimir en pantalla ("Intentos: " + contador + " | Costo Minimo: " + minCosto + " | Detalle Ruta: ")
```

```
viajero.imprimirRutaCostoMinimo(minRuta)
```

Algoritmo

chequearRuta

Entrada

```
Clase Viajero
      -  calcularCostoRutaActual
Algoritmo arrayCopy del Sistema
Secuencia ruta
Numero entero escala
```

Salida

Secuencia con la mejor ruta parcial

Pseudocódigo

```
Si (escala = viajero.cantCiudades)
      costo = viajero.calcularCostoRutaActual(ruta, true)
      Si (minCosto < 0 || costo < minCosto)
            minCosto = costo
            arraycopy(ruta, 0, minRuta, 0, ruta.length)
            devolver
      Fin si
Fin si
Bucle:
      Para i = 1 hasta viajero.cantCiudades
            Para j = 0 hasta escala
                  Si(ruta[j] = i)
                        continuar bucle
                  Fin si
            Fin para
```

```
        ruta[escala] = i;
        chequearRuta(ruta, escala + 1)
    Fin para
```

3.2.4. Análisis de complejidad del pseudocódigo anterior

La complejidad es de (N-1)!

3.2.5. Implementación en java del pseudocódigo propuesto

```java
import java.util.Random;

public class Viajero {

	public final static int MAP_SIZE = 200;
	private int[][] coordenadas;
	private int[][] costo;
	public int cantCiudades;
	boolean[] visitado;

	private static int[][] crearCoordenadasRandom(int n, Random random) {
		int[][] coordenadas = new int[n][2];
		for (int i = 0; i < coordenadas.length; i++) {
			// Se ignora el caso de 2 ciudades con las mismas coordenadas
			coordenadas[i][0] = Math.abs(random.nextInt() % MAP_SIZE);
			coordenadas[i][1] = Math.abs(random.nextInt() % MAP_SIZE);
		}
		return coordenadas;
	}
```

```
public Viajero(int n, Random random){
	this(crearCoordenadasRandom(n, random), n);
}

public Viajero(int[][] coordenadas, int n){
	this.cantCiudades = n;
	visitado = new boolean[n];
	costo = new int[n][n];
	this.coordenadas = coordenadas;
	costosPorCoordenadas();
}

// Se crea la matriz de costo al crear coordenadas para las ciudades y usando la distancia de vuelo como costo.
// Sin embargo, el algoritmo es mas general; matrices de costos arbitrarios deberian funcionar, incluso las asimetricas
private void costosPorCoordenadas() {
	for(int i = 0; i < coordenadas.length; i++) {
		for(int j = 0; j < coordenadas.length; j++) {
			costo[i][j] = calcularCostoDeViajeEntreCiudades(i, j);
		}
	}
}

private int calcularCostoDeViajeEntreCiudades(int i, int j) {
	int dx = coordenadas[i][0] - coordenadas[j][0];
	int dy = coordenadas[i][1] - coordenadas[j][1];
	return (int)Math.sqrt(dx * dx + dy * dy);    //Pitagoras
}

public double calcularCostoRutaActual(int[] ruta, boolean detallado) {
	double costoDeViaje = 0;
	for (int i = 1; i < ruta.length; i++) {
		costoDeViaje += costo[ruta[i-1]][ruta[i]];
		if (detallado) {
```

```
                        System.out.println("Costo desde Ciudad '" + ruta[i - 1] + "' hasta Ciudad '" + ruta[i] + "': " +
                        costo[ruta[i-1]][ruta[i]]);
                }
        }
        // volver a la ciudad de origen
        costoDeViaje += costo[ruta[cantCiudades-1]][ruta[0]];
        if(detallado){
                System.out.println("Costo desde Ciudad '"+ruta[cantCiudades - 1]+"' hasta Ciudad '" + ruta[0] + "': " +
                costo[ruta[cantCiudades-1]][ruta[0]]);
        }
        return costoDeViaje;
}

public void imprimirRutaCostoMinimo(int[] ruta) {
        for(int i = 0; i < ruta.length; i++){
                System.out.print(ruta[i] + " -> ");
        }
        System.out.println(ruta[0]);
}

public void imprimirMatrizDeCostosEntreCiudades() {
        System.out.println("Matriz de costo para el problema del viajante de comercio: ");
        System.out.print("<-> ");
        for(int i = 0; i < costo.length; i++){
                System.out.print(i + "    ");
        }
        System.out.println();
        for(int i = 0; i < costo.length; i++) {
                for(int j = 0; j < costo[i].length; j++) {
                        String espacios = "    ";
                        espacios = (costo[i][j] > 9) ? "   " : espacios;
                        espacios = (costo[i][j] > 99) ? "  " : espacios;
                        if(j==0)
                                System.out.print(i + "   ");
                        System.out.print(costo[i][j] + espacios);
```

```
                        }
                        System.out.print("\n");
                }
        }
}

public class ViajeroFuerzaBruta {

        private Viajero viajero;
        private double minCosto;
        private int[] minRuta;
        private long contador;

        public ViajeroFuerzaBruta(Viajero viajero){
                this.viajero = viajero;
        }

        public void EjecutarFuerzaBruta() {
                int[] ruta = new int[viajero.cantCiudades];
                minRuta = new int[viajero.cantCiudades];
                minCosto = -1;
                contador = 0;
                ruta[0] = 0; //la primera ciudad es siempre 0
                for(int i = 1; i < viajero.cantCiudades; i++){
                        ruta[1] = i;
                        chequearRuta(ruta, 2);
                }
                System.out.println("------------------------------------------------------------");
                System.out.println("Resultado de Fuerza Bruta:");
                System.out.println("------------------------------------------------------------");
                System.out.print("Intentos: " + contador + " | Costo Minimo: " + minCosto + " | Detalle Ruta: ");
                viajero.imprimirRutaCostoMinimo(minRuta);
        }

        private void chequearRuta(int[] ruta, int offset) {
```

```
		if(offset == viajero.cantCiudades){
			contador++;
			if(contador % 100000 == 0){
				System.out.println("Ruta Chequeada " + contador);
			}
			double costo = viajero.calcularCostoRutaActual(ruta, true);
			System.out.println();
			if(minCosto < 0 || costo < minCosto){
				minCosto = costo;
				System.arraycopy(ruta, 0, minRuta, 0, ruta.length);
			}
			return;
		}
		loop: for(int i = 1; i < viajero.cantCiudades; i++){
			for(int j = 0; j < offset; j++){
				if(ruta[j] == i){
					continue loop;
				}
			}
			ruta[offset] = i;
			chequearRuta(ruta, offset + 1);
		}
	}
}
```

4. Fuentes

- Guía de apuntes de la cátedra de Programación III
- http://www.ecured.cu/index.php/Floyd-Warshall
- http://queaprendemoshoy.com/problema-del-viajante/
- http://eio.usc.es/pub/mte/descargas/ProyectosFinMaster/Proyecto_762.pdf
- http://www.dc.uba.ar/materias/aed3/2013/1c/teorica/algo3_metah_heu.pdf
- http://www-eio.upc.es/~nasini/Blog/TSP_Notes.pdf
- http://www.academia.edu/3828405/IMPLEMENTATION_OF_HEURISTICS_FOR_SOLVING_TRAVELLING_SALESMAN_PROBLEM_USING_NEAREST_NEIGHBOURANDNEARESTINSERTIONAPPROACHES
- https://es.wikipedia.org/wiki/Problema_del_viajante