

# PRÁCTICA 4:

## EL PROBLEMA DEL VIAJANTE DE COMERCIO

ANÁLISIS DEL PROBLEMA / MANUAL DE USUARIO / ANEXO



David Ramírez Sierra  
[davidramirez@correo.ugr.es](mailto:davidramirez@correo.ugr.es)

## Table of Contents

<b>1. INTRODUCCIÓN .....</b>	<b>3</b>
El problema del viajante de comercio: .....	3
El vecino más cercano: .....	3
<b>2. ANÁLISIS DEL PROBLEMA .....</b>	<b>4</b>
Enunciado y diagrama UML: .....	4
Ciudad .....	4
Ruta .....	4
Problema.....	4
Heurística.....	4
Justificación de la solución: .....	5
Ciudad: .....	5
Ruta: .....	5
Problema: .....	5
Heurística: .....	6
Mejoras implementadas:.....	7
Soluciones que utilizan estrategias de inserción ( algo. 2 y algo. 3 ) .....	7
Soluciones con algoritmos que usan números aleatorios (algoritmo 4) .....	10
Análisis comparativo .....	11
<b>3. MANUAL DE USUARIO.....</b>	<b>12</b>
Ejemplos de ejecución del programa .....	13

# 1. INTRODUCCIÓN

## El problema del viajante de comercio:

El objetivo de esta práctica es que el alumno aplique los contenidos de los primeros temas de la asignatura (introducción a la programación dirigida a objetos, arrays y clases) para la resolución de un problema concreto: el problema del viajante de comercio (TSP, por Travelling Salesman Problem).

En términos sencillos, el TSP se define como sigue: dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la cantidad de kilómetros recorridos sea mínima. Más formalmente, dado un grafo  $G$ , conexo y ponderado, y dado uno de sus vértices  $v_0$ , encontrar el ciclo hamiltoniano de mínimo costo que comienza y termina en  $v_0$ .

Este problema es uno de los más relevantes en la clase de los NP-Complejos y encuentra aplicación práctica en herramientas relacionadas con transporte, logística y también en la industria electrónica. Por ejemplo, consideremos el brazo de un robot encargado de soldar las conexiones de un circuito integrado: el camino más corto que permite visitar cada punto a soldar, es el recorrido más eficiente para el robot. Una aplicación similar aparece cuando se desea minimizar el tiempo que utiliza un plotter para dibujar una figura.

Se denomina “instancia” del TSP a un conjunto particular de ciudades. Una solución (una “ruta”) para una instancia es una permutación del conjunto de ciudades que indica el orden en que se deben recorrer. En el cálculo de la longitud del recorrido no se debe olvidar sumar la distancia que existe entre la última ciudad y la primera; es decir, se debe cerrar el ciclo.

Por su interés teórico y práctico, existe una variedad muy amplia de algoritmos para abordar la solución del TSP y sus variantes. Siendo un problema NP-Completo, el diseño y aplicación de algoritmos exactos para su resolución no es factible en todos los casos. En la actualidad, se pueden resolver problemas de aproximadamente 15000 ciudades de forma exacta, pero para llegar a estos valores, se han requerido años de estudio en las áreas de investigación operativa, estructura de datos, arquitecturas paralelas de cómputo, etc.

En este guión, nos centraremos en una serie de algoritmos aproximados de tipo Greedy y evaluaremos su rendimiento en un conjunto de instancias del TSP. Por lo general estos algoritmos no encuentran la solución óptima del problema, pero son capaces de obtener soluciones razonablemente buenas en tiempo reducido.

## El vecino más cercano:

Se abordará la resolución del problema utilizando un método heurístico denominado “el vecino más cercano” cuyo funcionamiento es extremadamente simple: dada una ciudad inicial  $v_0$ , se agrega como ciudad siguiente aquella  $v_i$

(no incluida en la ruta) que se encuentre más cercana a  $v_0$ . El procedimiento se repite hasta que todas las ciudades se hayan incluido.

En la implementación solicitada, el método debe aplicarse desde cada posible ciudad inicial y el resultado final será el recorrido de menor longitud. Al final de la ejecución del programa, el programa debe haber calculado el orden en que se recorren las ciudades siguiendo la heurística del vecino más cercano. La solución obtenida podrá mostrarse en la salida estándar según se indica más adelante. El texto generado en la salida podría guardarse en un fichero de texto, y usarse para generar gráficos.

Para resolver el problema se propone la implementación de 4 clases cuyos nombres y funcionalidades básicas se describen a continuación.

## 2. ANÁLISIS DEL PROBLEMA

### Enunciado y diagrama UML:

#### Ciudad

Permite almacenar un par  $(x, y)$  (asociado a la posición de una ciudad). Cada ciudad tiene asociada una etiqueta (un número entero que va de 1 al número de ciudades). Además, dada una ciudad, tiene que permitir calcular la distancia euclídea a otra ciudad.

#### Ruta

Esta clase representa una solución del problema. La solución se define a través de una permutación de ciudades, o sea, un orden en el que se visitarán las ciudades. Debe permitir la posibilidad de representar soluciones parciales al problema, es decir cuando no todas las ciudades estén incluidas en la ruta todavía. La ciudad en el orden número 0 representa la ciudad de partida, la 1 la segunda en visitarse y así sucesivamente

#### Problema

Esta clase permitirá almacenar:

- El tamaño del problema
- La lista de ciudades
- La matriz de distancias entre ellas

La clase Problema debe disponer de un método que lea los datos de un problema de la entrada estándar.

#### Heurística

En esta clase se incluirán los métodos que resuelvan el problema, mediante el cálculo de un objeto Ruta. Dispondrá de un método que calcule un objeto Ruta con el resultado de resolverlo utilizando la heurística del vecino más cercano.

También tendrá un método para recuperar la solución calculada(objeto Ruta) por esta heurística.

La clase dispondrá también de un método que devuelva el coste de la solución obtenida.

## Diagrama UML de clases

### Justificación de la solución:

#### Ciudad:

Esta clase representa el elemento fundamental que constituye una ruta. Una ciudad tiene como datos miembro su coordenada y su etiqueta de identificación. La identificación es única para cada clase, por lo que no puede haber más de una ciudad con el mismo identificador. Las componentes de la coordenada son de tipo double y tienen la parte de abscisas (componente x) y la parte de ordenadas (componente y). Por tanto, para construir un objeto de tipo ciudad es necesario su coordenada x e y, y su número entero de identificación.

Por otro lado, esta clase incluye métodos públicos getter y setter, que son útiles para las demás clases. De este modo se pueden obtener o cambiar las coordenadas de una ciudad así como su identificador. Además, contiene un método llamado distanciaEuclídea que devuelve un dato de tipo double. Este método consiste en calcular y retornar la distancia euclídea entre la ciudad propia y las coordenadas que se le pasa correspondientes a otra ciudad.

Como se acaba de comentar, una ciudad representa el elemento fundamental de una ruta, por tanto, un conjunto de ciudades forman una ruta. Sin embargo, no tiene sentido que una ciudad contenga una ruta.

#### Ruta:

La clase Ruta representa un conjunto de ciudades con un cierto orden y un determinado coste. Por tanto, los datos miembro de esta clase son un array de ciudades, un array de enteros que son las etiquetas, el coste de la ruta y el número de ciudades que se encuentran en la ruta en ese determinado instante.

Para construir un objeto de tipo Ruta, tan solo es necesario pasarle el número de ciudades. Como métodos públicos, además de los setter y getter, tenemos un método booleano llamado anadirCiudadARuta que se encarga de añadir en la última posición libre el objeto de tipo Ciudad que se le pasa. De manera que cuando no sea posible añadir una ciudad a la ruta, devuelva false.

#### Problema:

La clase Problema se encarga de recoger el problema. Se trata de leer y almacenar los datos de las diferentes ciudades que presentan el problema. Por tanto, los datos miembros de esta clase son el número de ciudades y el array de ciudades que conforman el problema.

Así mismo, cabe esperar que para construir un objeto de tipo Problema, no es necesario pasarle ningún parámetro. Además de los métodos públicos setter y getter, tiene un método público imprescindible que se encarga de leer los datos problema. Esta lectura se hace a partir de la entrada estándar, leyéndolo de un fichero de texto (.txt o .tsp) y que tiene un formato específico. El método consiste en leer en primer lugar el tamaño del problema (esto es, la dimensión) y en segundo lugar, con ayuda de un bucle for, recoger el identificador y las coordenadas de cada ciudad. Todo ello se almacena como corresponde en el array de ciudades miembro de la clase Problema.

### Heurística:

Esta clase es la responsable de resolver el problema. De manera que tiene como datos miembro la ruta solución y el problema. Por tanto, para construir un objeto de tipo Heurística, es necesario pasarle como parámetro un objeto de tipo Problema. Por otro lado, vemos que la relación entre problema y heurística es 1 a 1. Es decir, para cada objeto Heurística, le corresponde un problema el cual ha de resolver.

Como se comentaba anteriormente, la clase Heurística es capaz de resolver el problema siguiendo varios algoritmos. En esta sección se explicará tan solo el algoritmo del vecino más cercano. Los demás algoritmos están expuestos en la parte dedicada a las mejoras implementadas.

### *Pseudo-código del algoritmo (método public void vecinoMasCercano())*

Como se expuso en la parte de introducción, el algoritmo del vecino más cercano consiste en: dada una ciudad inicial  $v_0$ , se agrega como ciudad siguiente aquella  $v_i$  (no incluida en la ruta) que se encuentre más cercana a  $v_0$ . Esto se realiza hasta que todas las ciudades hayan sido incluidas. Además, en la implementación solicitada, el método debe aplicarse desde cada posible ciudad inicial, siendo el resultado final el recorrido de menor longitud.

A continuación, se explica grosso modo en qué consiste mi algoritmo del vecino más cercano:

#### INICIO

```
/*Partimos de dos listas ( ciudadesLibres y ciudadesOcupadas)*/
Lista ciudadesLibres = problema.getArrayDeCiudades(); //Es decir,
cogemos las ciudades a ordenar
Lista ciudadesOcupadas = vacio; //Se trata de una lista auxiliar que se va
rellenando a medida que vamos ordenando

FOR (i = 0; i<problema.getNumeroCiudades; i++) //hacemos un bucle para
recorrer todas las ciudades, ya que , para cada instancia de inicio obtenemos una ruta

    escogemos una instancia(i); //Sería la ciudad de partida

    MIENTRAS ( haya ciudades libres)

        buscamos, para cada ciudad, la ciudad libre más
        cercana;

        dicha ciudad, la añadimos a las ciudadesOcupadas y la
        eliminamos de las ciudadesLibres;

    FIN MIENTRAS

    coste = calcularCoste(ciudadesOcupadas); //Con un método privado
calculamos el coste de la ruta auxiliar ordenada. Cabe recalcar que el coste
también se podría haber obtenido como la suma total de las distancias menores
parciales que se dan en el bucle Mientras

    SI ( coste < coste_menor_encontrado_hasta_el_momento)

        rutaFinal = ciudadesOcupadas; // nos quedamos con esa ruta

    FIN SI

    limpiamos la lista auxiliar de ciudadesOcupadas;
```

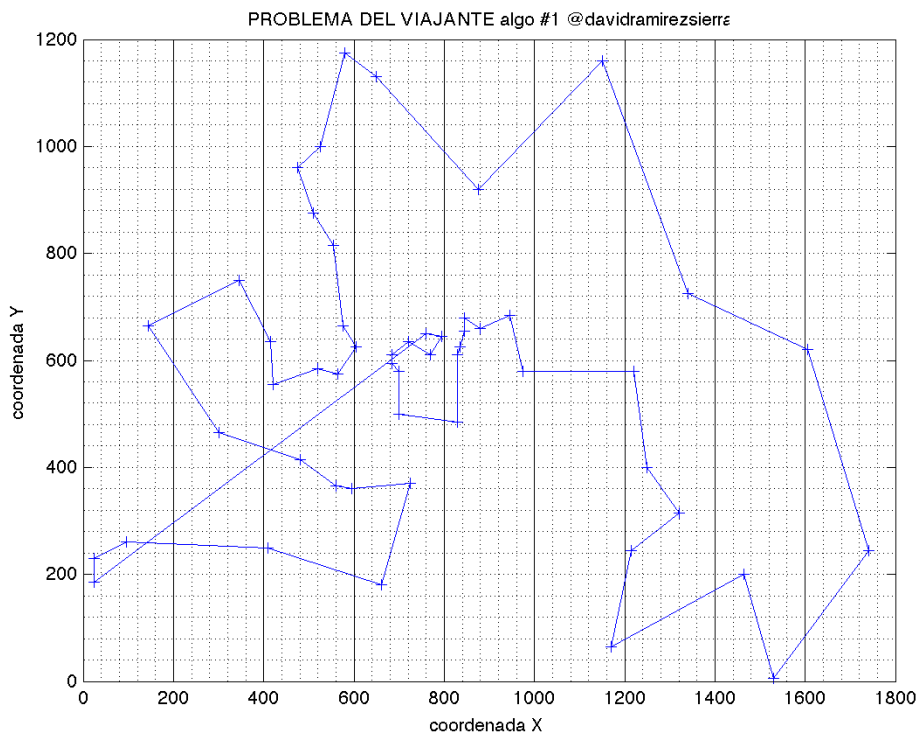
```
ciudadesLibres = problema.getArrayDeCiudades(); //Inicializamos  
el problema
```

FIN FOR  
FIN

Tras realizar el algoritmo, guardamos la ruta final en el dato miembro del objeto problema añadiendo las ciudades con el método correspondiente [anadirCiudadARuta(Ciudad ciudad)], el cual ya crea la rutaSolución. Creo que es más eficiente el uso de rutas auxiliares y finalmente modificar la rutaSolución, que en cada bucle ir añadiendo o eliminando ciudades de dicha ruta.

Como sabemos, esta ruta no es la más óptima debido a que el coste no es el óptimo. Aún así, ofrece resultados agradables, sobretodo en problemas de tamaño reducido, donde no hay muchas posibilidades de caminos diferentes.

Un ejemplo de solución usando el algoritmo del vecino más cercano, para 52 ciudades, es el siguiente:



Para esta ruta solución, obtenemos un coste de 8182.19

### Mejoras implementadas:

En esta sección se explica la implementación de las mejoras que se proponen como tarea opcional. En concreto, en mi trabajo, he incluido los algoritmo 2, 3 y 4, así como la realización de un análisis de diferentes aspectos de dichos algoritmos. De manera que he dejado el algoritmo 5 (Monte Carlo), para completarlo en un futuro.

### Soluciones que utilizan estrategias de inserción ( algo. 2 y algo. 3 )

Este tipo de estrategias parten de un recorrido inicial, que en mi caso, he realizado mediante el triángulo exterior más grande. Es decir, tomando como vértices la ciudad más a la derecha, más a la izquierda y más arriba.

A partir del recorrido inicial, se van insertando en la posición correspondiente, las ciudades hasta completar la ruta final. Por tanto, es el criterio de elegir la

ciudad a insertar el que marca la diferencia entre los algoritmos 2 y 3. De manera que hay dos posibilidades, la inserción más económica y la inserción del más lejano:

### *Inserción Más Económica (algoritmo 2)*

De entre todas las ciudades no visitadas, elijo aquella que provoque el mínimo incremento de la longitud. Este es un enfoque puramente greedy. O sea, el funcionamiento sería el siguiente:

INICIO

// Parto del recorrido triangular

MIENTRAS (ciudadesLibres no esté vacía)

FOR (i=0; i<ciudadesLibres.tamaño(); i++)

ciudad\_aux = ciudadLibre(i) // probamos con todas las ciudades no añadidas a la ruta que queden

FOR (j=0; j<ciudadesOcupadas-1; i++)

dist = d1 + d2 - d; // Esto se explica en la figura(\*)

Reservamos la menor distancia y la posición de donde se encuentra dicha ciudad;

FIN FOR

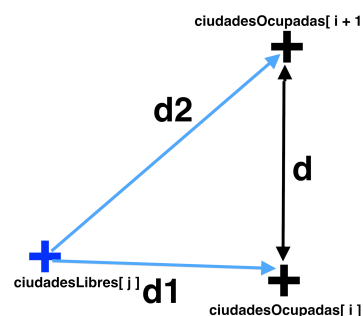
Reservamos la ciudad que supone el menor coste al ser añadida a la ruta;

FIN FOR

Añadimos la ciudad que incrementa el menor coste en la posición que le corresponde y la eliminamos de las ciudadesLibres;.

FIN MIENTRAS

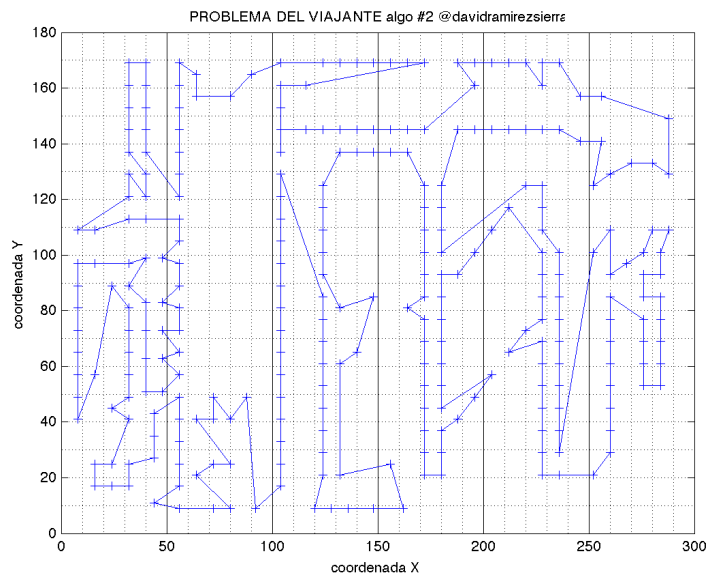
FIN



Cabe destacar que, cada vez que hago referencia a reservar el menor elemento, estoy suponiendo que todo ello se hace a través de un "if" en donde interviene una variable que es la que se ha calculado y otra variable auxiliar que se encarga de ir almacenando cada vez que haya un elemento menor .



Un ejemplo de solución para 280 ciudades, usando este algoritmo, es el siguiente:

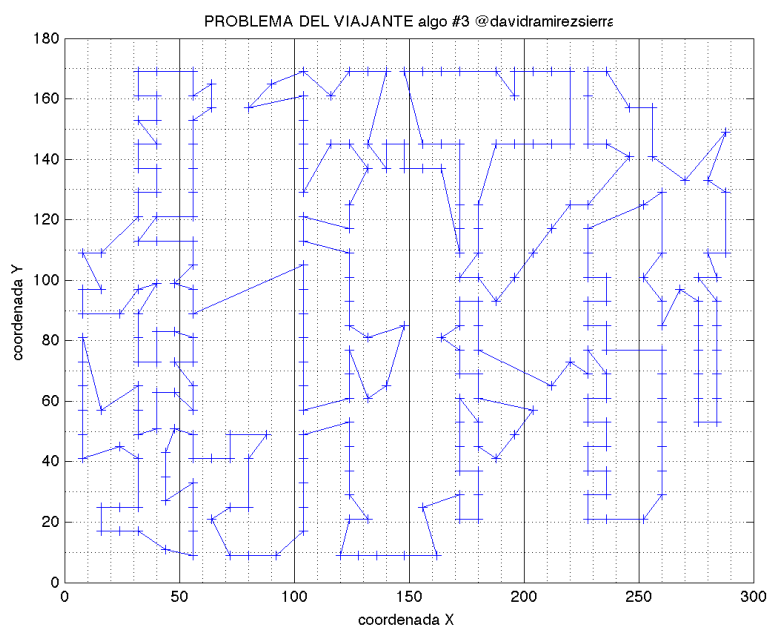


Obtenemos un coste total de 3042.32

### *Inserción del Más Lejano (algoritmo 3)*

En este caso, se trata de, entre todas las ciudades no visitadas, insertar aquella cuya distancia mínima a cualquier ciudad visitada, sea máxima. La idea de este enfoque es fijar la distribución general de la ruta en los primeros pasos de la construcción. No puedo comentar mucho en cuanto a la implementación de esta tarea, puesto que es muy similar al apartado anterior. Tan solo se diferencia el hecho de que el criterio de elección se basa en escoger aquella ciudad que su distancia mínima sea la máxima.

Aunque a priori no lo parezca, este algoritmo es el que me ha dado mejores resultados. Aunque, este aspecto se tratará en el apartado de análisis.



### Soluciones con algoritmos que usan números aleatorios (algoritmo 4)

En este apartado se propone implementar un algoritmo que genere K permutaciones (rutas) aleatorias del conjunto de ciudades y devuelva el mínimo valor encontrado. Para ello asumimos  $K = 10000$ .

INICIO

FOR (k=0; k<K ; k++)

    ciudadesLibres = problema.getArrayDeCiudades();

    MIENTRAS (ciudades libres no esté vacía)

        generamos un número aleatorio entero entre 0 y el número de ciudades no visitadas (ciudadesLibres);

        tomamos ese número como la posición de donde coger la ciudad de entre las ciudades no visitadas y lo añadimos a la ruta;

        además, eliminamos esa ciudad de los no visitados (ciudadesLibres)

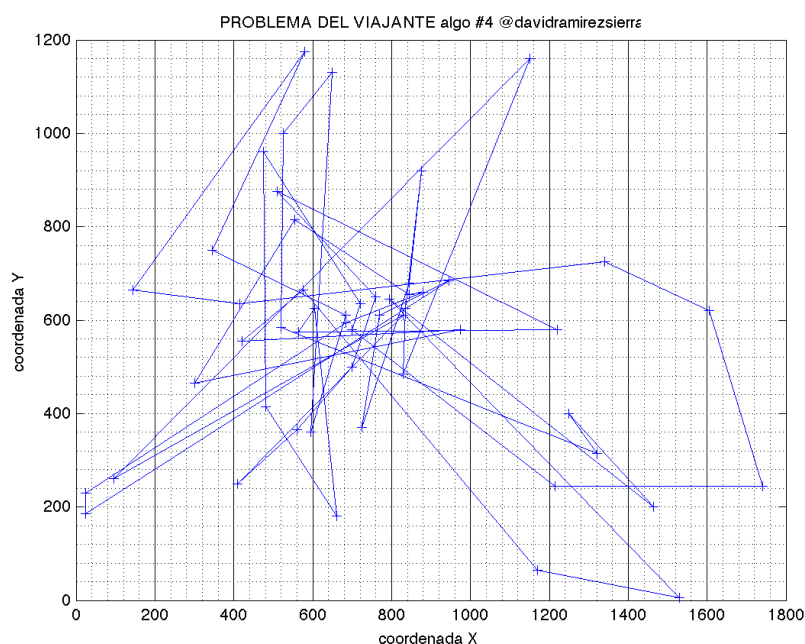
        Reservamos el coste para esa permutación. Guardamos la ruta que por ahora nos está dando el menor coste

    FIN MIENTRAS

FIN FOR

FIN

Como cabe esperar, este sistema no es muy bueno. Puede dar un buen resultado para rutas de poco tamaño, es decir, para problemas sencillos. Pero si el tamaño es muy grande, el resultado deja mucho que desear. A continuación se muestra un ejemplo para un total de 52 ciudades.



### Análisis comparativo

En esta tarea, se evalúa el rendimiento de las estrategias que se han implementado sobre el conjunto de instancias del TSP disponibles en la plataforma docente, para las cuales se conoce la longitud del recorrido óptimo. Para ello, tomaremos cada instancia del conjunto y la resolveremos con cada algoritmo, registrando en una tabla los costos encontrados.

Para la realización de esta tarea se ha implementado una nueva clase main llamada ANALISIS.java en la cual se crea un objeto de tipo problema. En ella se encuentra un sencillo código que, a partir de un fichero de entrada .tsp, solicita solucionar cada problema y mide, para cada algoritmo, diferentes parámetros.

En concreto, resuelve el problema según 4 algoritmos diferentes. Genera un fichero de texto de salida donde se informa de todos los parámetros:

Un ejemplo de fichero de texto de salida para el fichero de entrada a280.tsp es el siguiente:

```
analisis_a280.txt
1 REALIZANDO ANALISIS
2 APLICANDO ALGORITMO 1 vmc2...
3 COSTE: 3094.278434464387
4 opt%: 19.97977644297739 mili:198
5 APLICANDO ALGORITMO 2 ime...
6 COSTE: 3042.319520242773
7 opt%: 17.965084150553423 mili:112
8 APLICANDO ALGORITMO 3 iml...
9 COSTE: 2993.9173885307914
10 opt%: 16.088305100069462 mili:119
11 APLICANDO ALGORITMO 4 rnd...
12 COSTE: 30327.128334212775
13 opt%: 1075.9258756965016 mili:839
14
```

Si aplicamos este análisis para cuatro problemas de diferentes tamaños, obtenemos la siguiente tabla:

TSP	Optimo	Algo 1			Algo 2			Algo 3			Algo 4		
		min	%	mili	min	%	mili	min	%	mili	min	%	mi
small10	-		-			-						-	
berlin52	7542	8182.19	8.48	32	8286.86	9.87	19	8330.63	10.45	19	23630.85	213.32	21.
a280	2579	3094.27	19.97	198	3042.31	17.96	112	2993.91	16.088	119	30327.12	1075.92	83'
pr1002	259045	312237.26	20.53	5980	296508.41	14.46	4419	294302.99	13.61	4626	6074840.65	2245.0	344
Promedio			16.32%			14.09%			13.35%			1177.7%	

Como vemos en la tabla, el algoritmo que mejor resultados ofrece es el algoritmo que corresponde a la inserción del más lejano. Por otro lado, como era de esperar, el que ofrece los peores resultados es el algoritmo 4, basado en números aleatorios.

### 3. MANUAL DE USUARIO

Como sabemos, el programa consta de 5 clases, siendo una de ellas el programa principal llamado TSP.java . Además, hay una clase principal destinada a hacer el análisis comparativo llamada ANALISIS.java.

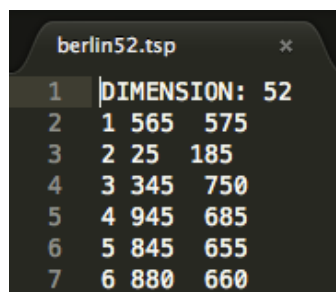
El comando para compilar el programa en el terminal es el siguiente:

```
Ciudad.java; javac Ruta.java; javac Problema.java; javac Heuristica.java; javac TSP.java; javac ANALISIS.java
```

Como se pedía en el enunciado, para la ejecución del programa hay que introducir los siguientes parámetros:

```
java TSP algo # solucion ruta coste <rutaProblema.tsp> salidaSolucion.txt
```

En “rutaProblema.tsp” se encuentran la ruta del problema escrita en texto plano y con el formato correspondiente, es decir:



1	DIMENSION: 52		
2	1	565	575
3	2	25	185
4	3	345	750
5	4	945	685
6	5	845	655
7	6	880	660

Dentro de los parámetros, se puede sustituir “# “ por 1, 2, 3 y 4 dependiendo del algoritmo que se desee utilizar. Además, si lo que se desea es una futura representación de la salida, es suficiente con escribir solo “solucion”, prescindiendo del parámetro “ruta” y “coste”. Si por el contrario tan solo se desea saber el orden de las ciudades a través de sus etiquetas, basta con poner solo ruta.

Para la representación gráfica y la generación de una imagen con una extensión .png, se hace a través de un fichero .m de Matlab. Al ejecutar este sencillo programa, has de elegir el fichero de salida que desees y, a través de un plot, te lo convertirá a .png.

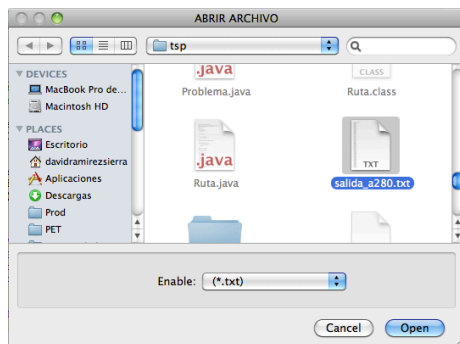
## Ejemplos de ejecución del programa

En primer lugar, compilamos todas las clases:

```
cvih217209:~ davidramirezsierra$ cd /Users/davidramirezsierra/Desktop/tsp javac Ciudad.java; javac Ruta.java; javac Problema.java; javac Heuristica.java; javac TSP.java; javac ANALISIS.java
```

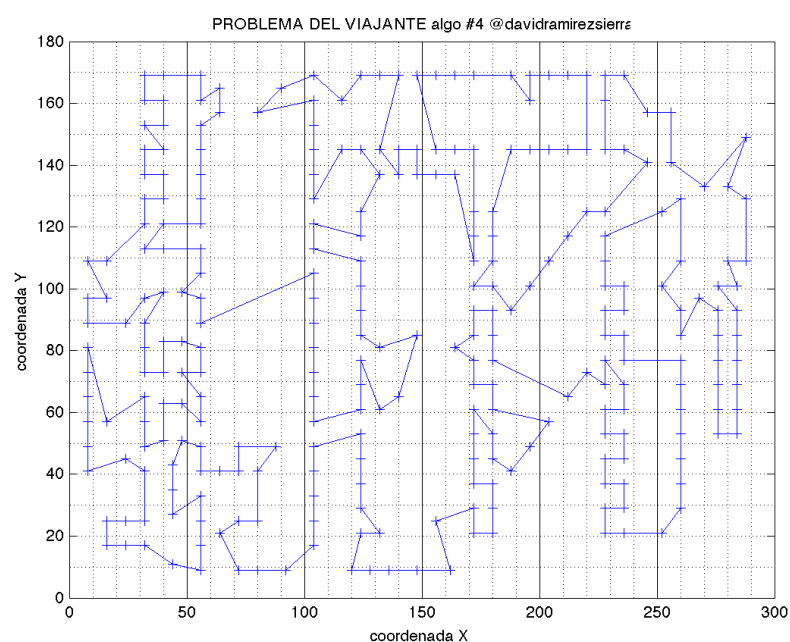
Una vez compilado correctamente, ejecutamos el comando para el fichero a280.tsp y que resuelva el problema con el algoritmo 3. Dicha solución la escribe en el fichero salida\_a280.txt, tal y como se indica en la siguiente figura:

```
cvih217209:tsp davidramirezsierra$ java TSP algo 3 solucion <a280.tsp>salida_a280.txt
cvih217209:tsp davidramirezsierra$ ls
ANALISIS.class      Problema.class      berlin52            pr1002.tsp
ANALISIS.java       Problema.java       berlin52.tsp        salida_a280.txt
Ciudad.class        Ruta.class          compilar.txt        small10
Ciudad.java         Ruta.java           dist.png            small10.tsp
Heuristica.class    TSP.class           eil101.tsp          txt2png.m
Heuristica.java     TSP.java            informe.docx
KROA200.tsp         a280                lin318.tsp
OPTIMOS.txt         a280.tsp            pr1002
```



Vemos cómo se ha generado un archivo .txt llamado salida\_a280.txt y que en su interior contiene la solución.

En el siguiente paso, vamos a representar los datos de la solución haciendo uso del script de Matlab. El archivo se llama "txt2png.m" ya que convierte de texto plano a formato .png. Tras ejecutar el script, seleccionamos el archivo que queremos representar y se generará una imagen con dicho formato:



Si, por ejemplo, tan solo se quiere obtener el coste de la ruta. Solamente ha de poner “coste” como parámetro. Tal y como indica la siguiente figura:

```
cvihs217209:tsp davidramirezsierra$ java TSP algo 3 coste <a280.tsp
APLICANDO ALGORITMO 3 iml...
COSTE: 2993.9173885307914
```

Otro modo de ejecución sería que tan solo te mostrara las etiquetas de la ruta solución en orden, para ello se escribiría el parámetro “ruta”:

```
cvihs217209:tsp davidramirezsierra$ java TSP algo 3 ruta <a280.tsp
APLICANDO ALGORITMO 3 iml...
RUTA (labels):
69
68
57
56
45
46
55
54
47
53
```

Para finalizar, para hacer un análisis de todos los algoritmos se ha de ejecutar el siguiente “java ANALISIS coste\_optimo <a280.tsp” :

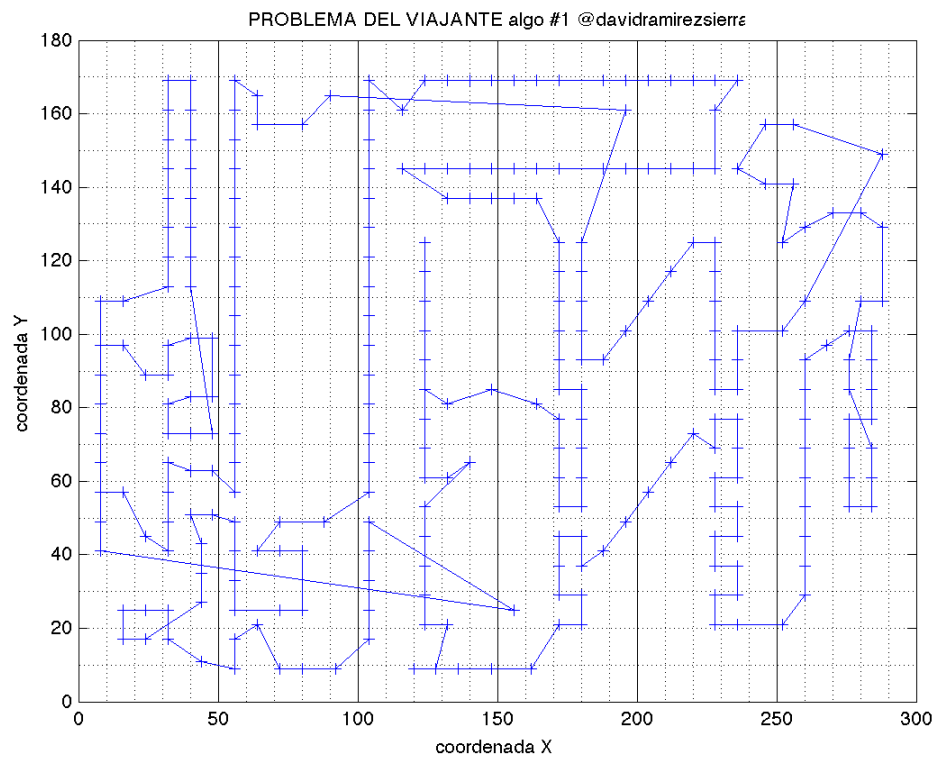
```
cvihs217209:tsp davidramirezsierra$ java ANALISIS 2579 <a280.tsp
REALIZANDO ANALISIS
APLICANDO ALGORITMO 1 vmc2...
COSTE: 3094.278434464387
opt%: 19.97977644297739 mili:195
APLICANDO ALGORITMO 2 ime...
COSTE: 3042.319520242773
opt%: 17.965084150553423 mili:112
APLICANDO ALGORITMO 3 iml...
COSTE: 2993.9173885307914
opt%: 16.088305100069462 mili:118
APLICANDO ALGORITMO 4 rnd...
COSTE: 30393.260748771932
opt%: 1078.4901414801059 mili:827
```

## ANEXO

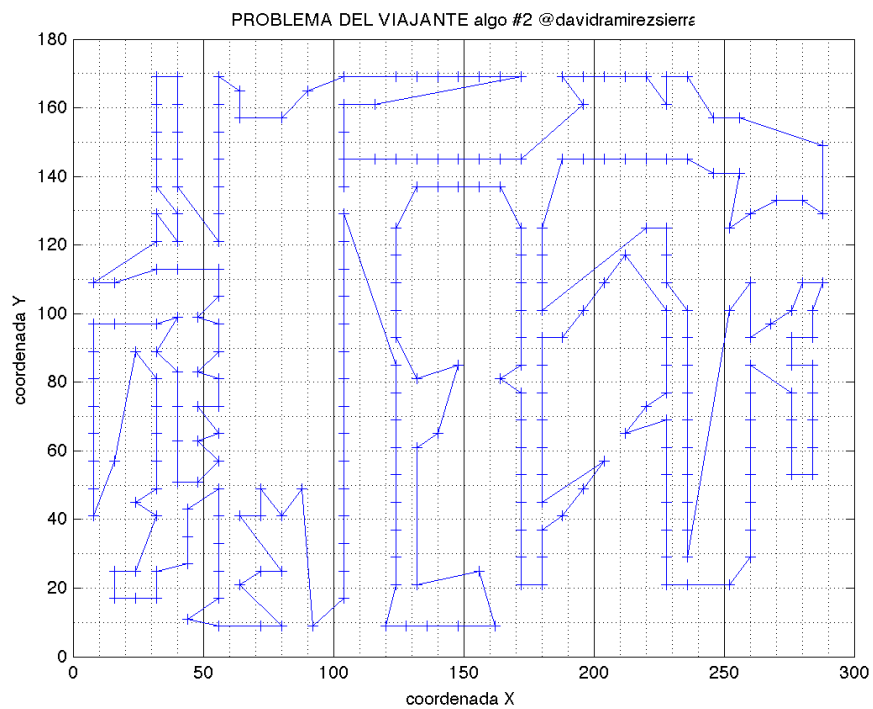
Resultados y gráficas obtenidas para diferentes ficheros .tsp:

a280.tsp.

Algoritmo 1 (comienza en la 161)

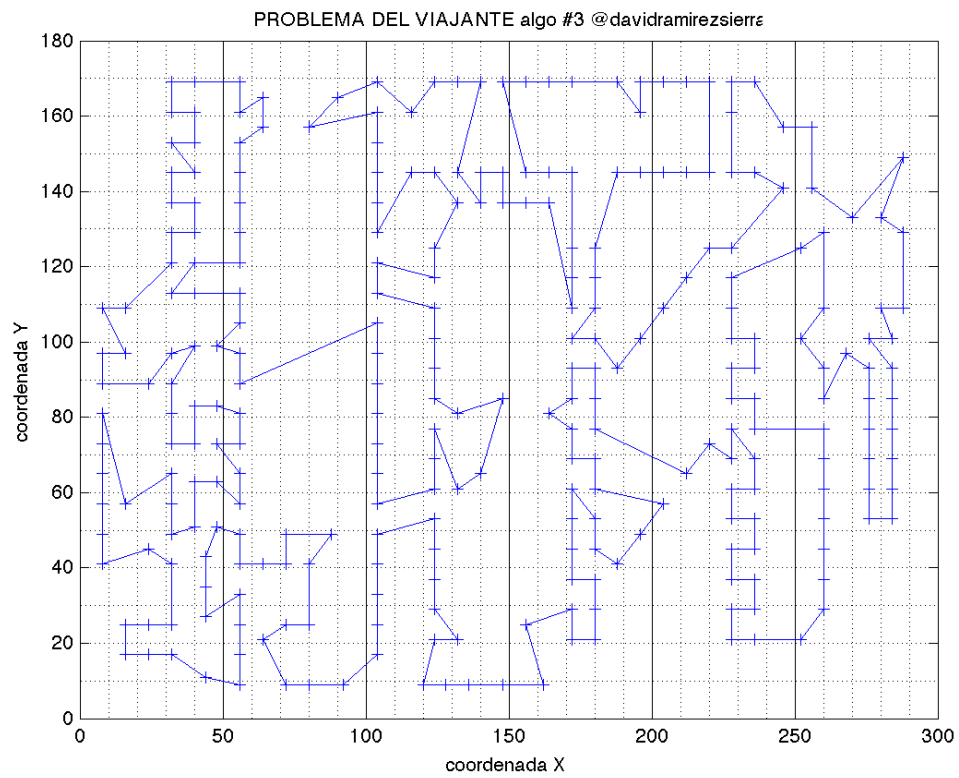


Algoritmo 2 (Comienza en la ciudad 69)

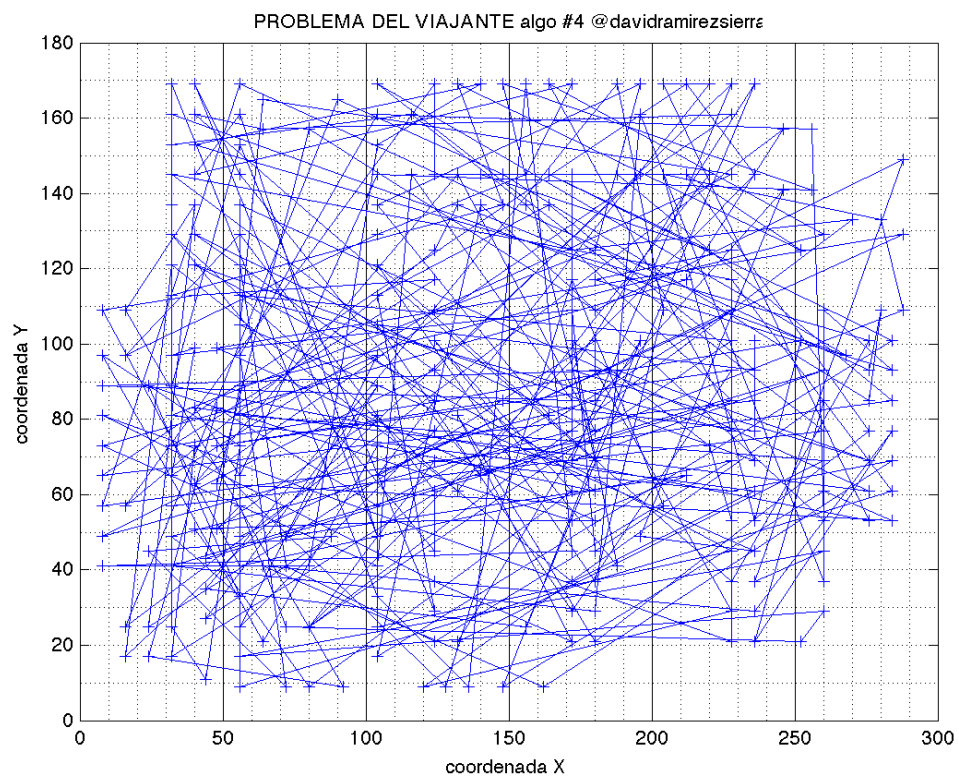




### Algoritmo 3 (comienza en la 69)

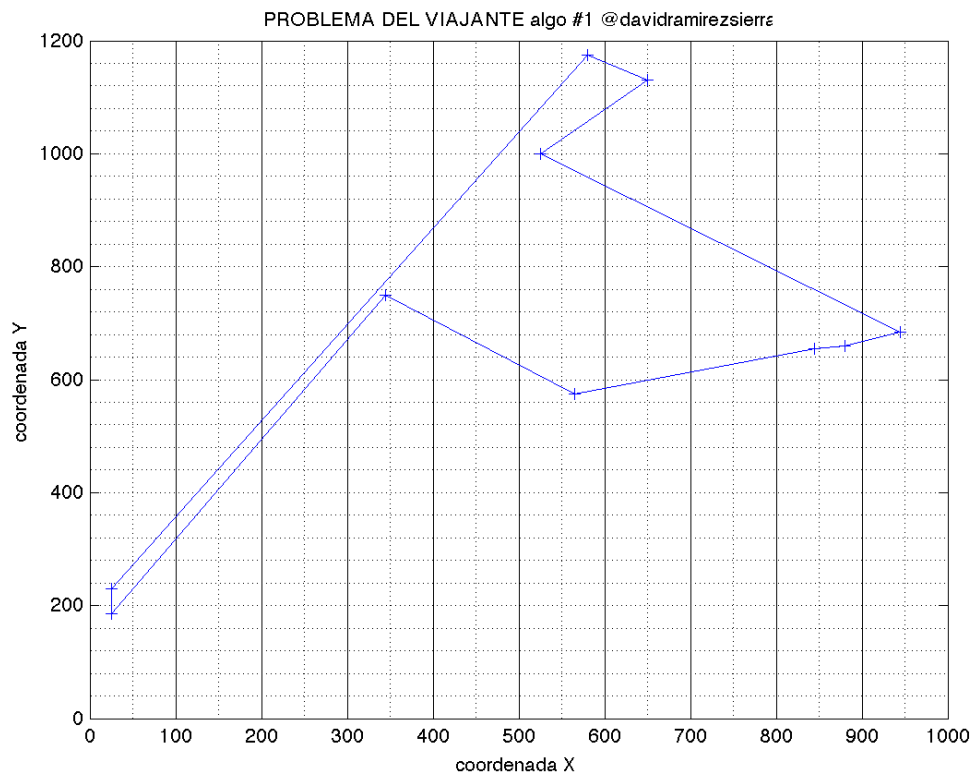


### Algoritmo 4 (Comienza en la 39)

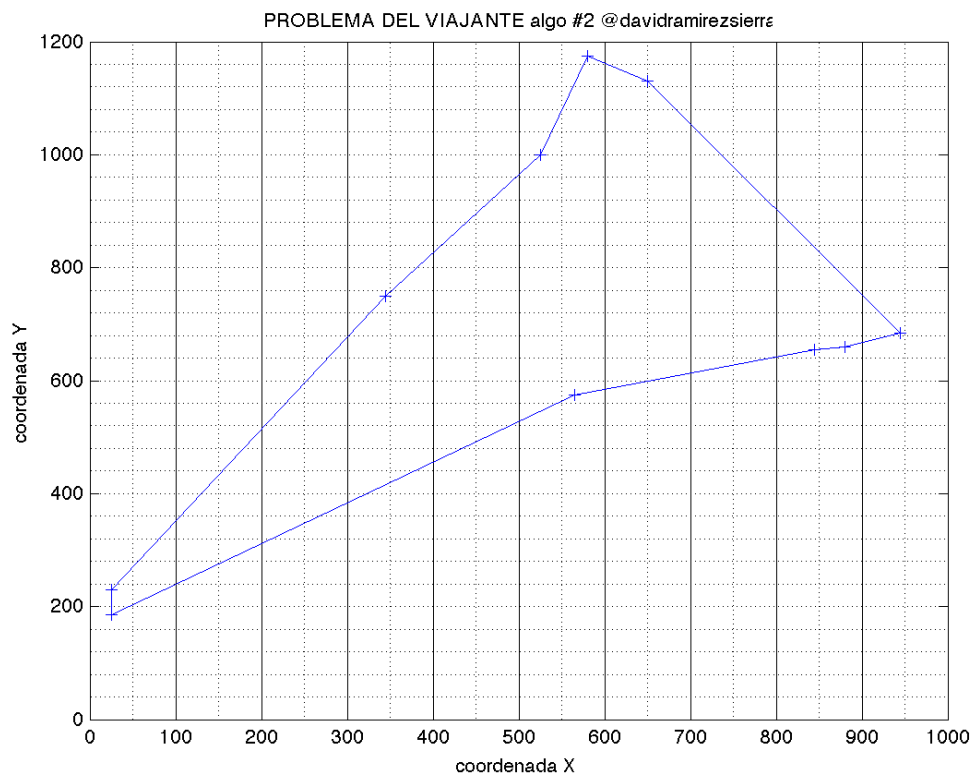


small10:

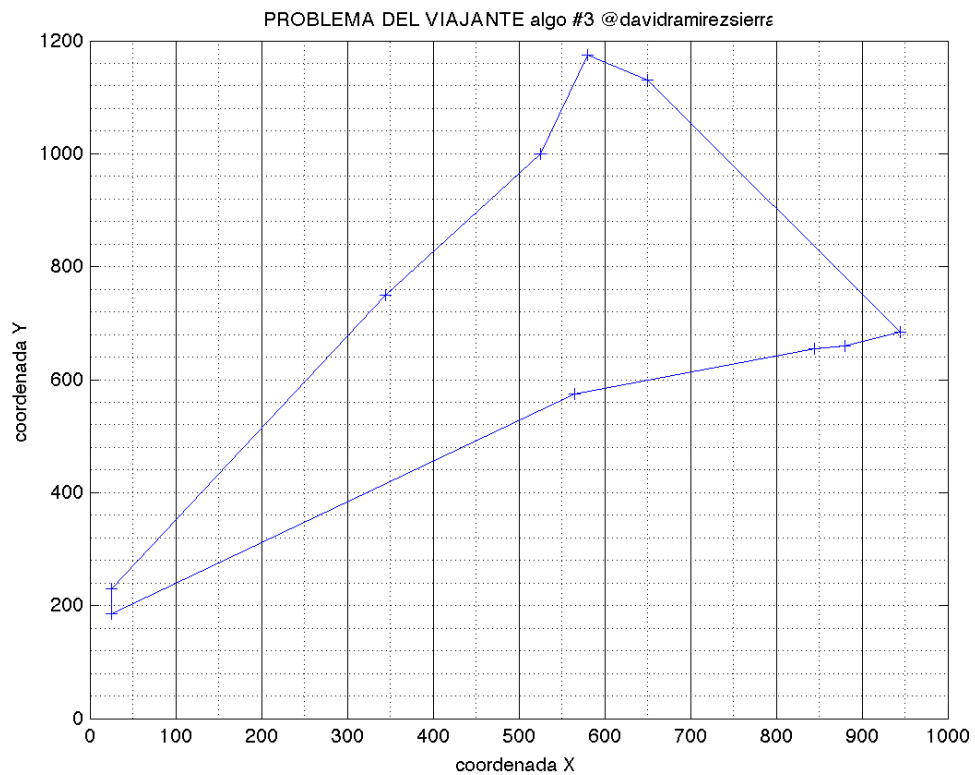
Algoritmo 1 (Comienza en la ciudad 3)



Algoritmo 2 (Comienza en la ciudad 2)



### Algoritmo 3 (Comienza en la ciudad 2)



### Algoritmo 4 (Comienza en la ciudad 7)

