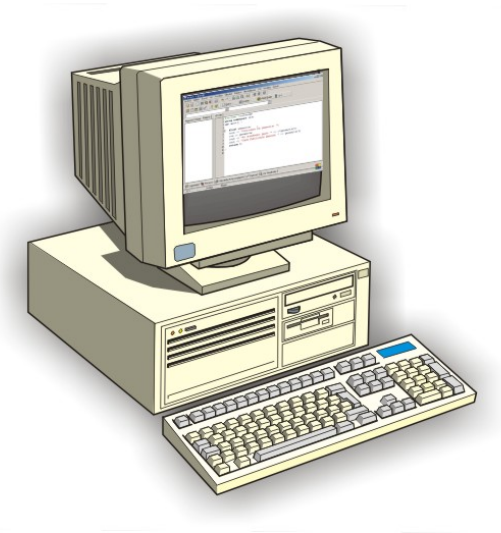


Práctica 4

Práctica evaluable: *El problema del viajante de comercio*

Marzo de 2015

DECSAI

Complementos de Programación

Curso 2014/2015

Contents

1 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[1] y ponderado[2], y dado uno de sus vertices v_0, encontrar el ciclo hamiltoniano[3] de mínimo costo que comienza y termina en v_0.

Este problema es uno de los más relevantes en la clase de los NP-Completos 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

[1] Un grafo G se dice conexo si, para cualquier par de vértices a y b en G, existe al menos una trayectoria (una sucesión de vértices adyacentes que no repita vértices) de a a b.

[2] Un grafo ponderado asocia un valor o peso a cada arista en el grafo. El peso de un camino en un grafo con pesos es la suma de los pesos de todas las aristas atravesadas.

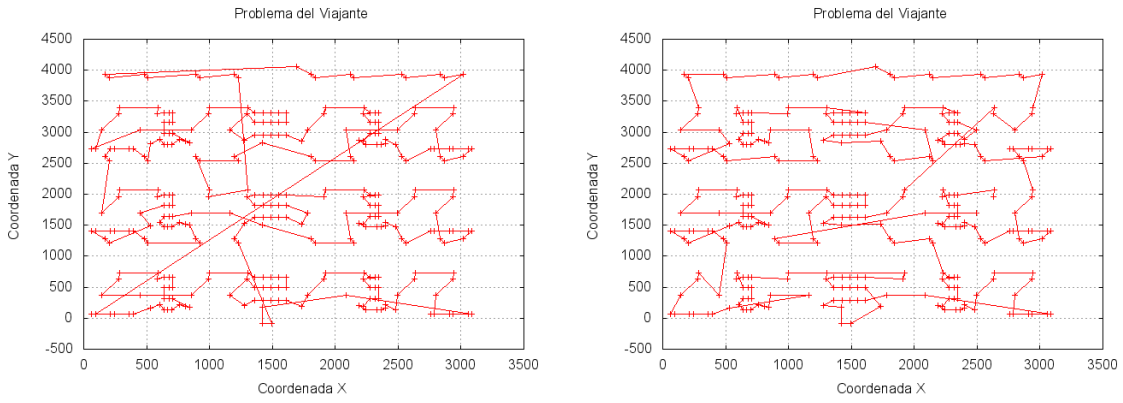
[3] Un camino hamiltoniano, en el campo matemático de la teoría de grafos, es un camino de un grafo, una sucesión de aristas adyacentes, que visita todos los vértices del grafo una sola vez. Si además el último vértice visitado es adyacente al primero, el camino es un ciclo hamiltoniano.

óptima del problema, pero son capaces de obtener soluciones razonablemente buenas en tiempo reducido.

2 Tareas

En este guion, 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.

La longitud del recorrido depende de la ciudad inicial elegida. En la Fig.1 se pueden ver los resultados de la aplicación de la heurística partiendo desde 2 ciudades diferentes.



Origen: 1
Coste: 54031

Origen: 193
Coste: 49201

Figure 1: Soluciones obtenidas con la heurística del vecino más cercano desde distintas ciudades de partida.

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 como los de la figura 1 que permiten visualizar las rutas.

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

Clase 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.

Clase 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.

Clase Problema

Esta clase permitirá almacenar:

- El tamaño del problema (número de ciudades).
- La lista de ciudades del problema.
- La matriz de distancias entre ellas.

La matriz de distancias se rellenará usando la distancia euclidea (ver más adelante cómo se calcula) entre ciudades.

A lo largo del programa, supondremos que las ciudades tienen asociado un número que va desde 0 hasta el número de ciudades menos 1. Para facilitar la programación del algoritmo de solución, se recomienda añadir un método que dato un número de ciudad n nos devuelva el objeto `Ciudad` correspondiente a ese número.

La clase `Problema` debe disponer de un método que lea los datos de un problema de la entrada estándar. Para facilitar la introducción de los datos por la entrada estándar, en la plataforma docente hay disponibles algunas instancias de ejemplo almacenadas en ficheros de prueba. El formato de los ficheros es el siguiente:

```
DIMENSION: 52
1 565.0 575.0
2 25.0 185.0
3 345.0 750.0
.............
.............
51 1340.0 725.0
52 1740.0 245.0
```

La primera fila indica la cantidad de ciudades en el fichero. Luego, aparece una fila por cada ciudad conteniendo tres valores: etiqueta de la ciudad (note que esta etiqueta no tiene nada que ver con el número entero que nosotros le asignaremos a cada ciudad según se ha indicado más arriba), la coordenada X y la coordenada Y. Para leer estos ficheros, no utilizaremos operaciones de entrada/salida de ficheros, sino métodos para leer de la entrada estándar, al igual que hicimos en las prácticas 1 a 3. En un programa Java, podemos leer datos de un fichero de texto (por ejemplo `ficheroprueba.txt`) usando órdenes para leer de la entrada estándar, si ejecutamos el programa de la siguiente forma:

```
java Programa < ficheroprueba.txt
```

Esto hará que el programa lea los datos del fichero `ficheroprueba.txt`, conforme los va solicitando, en lugar de pedirlos del teclado. Al leer las ciudades, el programa debe asociar automáticamente un número de ciudad a cada una de ellas usando sucesivamente los enteros $0, \ldots n-1$ (siendo n el número de ciudades).

Clase Heuristica

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.

Clase del main

Además de las clases anteriores, construiremos la clase donde estará incluida la función `main()`. Llamaremos `TSP` a esta clase. Al ejecutar el programa, éste pedirá los datos del problema por la entrada estándar, según se ha descrito anteriormente. A continuación calculará una solución con la heurística del vecino más cercano. Los datos que mostrará en la salida estándar dependerán de los parámetros con que se haya ejecutado la función `main()`:

- Parámetro **solucion**: mostrar las coordenadas de cada ciudad en el orden dado por la solución encontrada, según se ha mostrado en la sección anterior.

  ```
  java TSP solucion < ficheroprueba.txt
  ```

  Esto haría que apareciese en la salida estándar, algo así:

  ```
  2906 1400
  2937 1400
  3055 1400
  3087 1400
  3016 1276
  ...
  ```

- Parámetro **ruta**: mostrar las etiquetas de las ciudades en el orden en que se visitarán.

  ```
  java TSP ruta < ficheroprueba.txt
  ```

  Aparecería en la salida estándar algo así:

```
193
185
1
3
...
```

- Parámetro **coste**: mostrar el coste de la solución encontrada.

  ```
  java TSP coste < ficheroprueba.txt
  ```

  Esto haría que apareciese en la salida estándar, el coste de la mejor solución:

  ```
  MEJOR SOLUCIÓN: 628
  ```

- Se podrán usar simultáneamente varios parámetros. Por ejemplo usando:

  ```
  java TSP solucion coste < ficheroprueba.txt
  ```

  se mostraría:

  ```
  2906 1400
  2937 1400
  3055 1400
  3087 1400
  3016 1276
  ...
  MEJOR SOLUCIÓN: 628
  ```

3 Recomendaciones

Antes de empezar a implementar se debe pensar y analizar detenidamente el problema para poder identificar los métodos y datos miembro de cada clase. Para cada método identifique: qué datos necesita para trabajar (argumentos), qué devuelve (tipo de salida), cómo debe llamarse para que sea legible el código y finalmente qué debe hacer y cómo debe hacerlo. Esta última parte debe plasmarse en un pequeño esquema en papel, que debe realizarse antes de escribir línea alguna de código. Respecto a los datos miembro, decida los que necesitará en cada clase. Si fuera necesario, puede definir métodos auxiliares.

Se recomienda dibujar el diagrama de clases en UML antes de empezar a implementar cualquier código. Para ello puede emplearse por ejemplo el programa `dia`.

Se recomienda que los datos miembro sean siempre privados. Se harán públicos únicamente aquellos métodos que se necesiten usar en otras clases. El resto de métodos auxiliares se harán también privados.

4 Distancia entre ciudades

En este programa, supondremos que las distancias entre cada par de ciudades se calcula usando la distancia euclidea, con lo que la matriz de distancias del la clase `Problema` deberá rellenarse de esta forma. Para el cálculo correcto de la distancia entre dos ciudades debe proceder de la siguiente manera: sean (x_i, y_i) y (x_j, y_j) las coordenadas en el plano de las ciudades c_i y c_j respectivamente. La distancia euclídea entre ambas ciudades se obtiene de la siguiente forma:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2)} \tag{1}$$

5 Visualización gráfica de la solución

Podemos usar el programa `gnuplot` para visualizar la ruta encontrada por nuestro programa. Para ello guardaremos la solución encontrada en un fichero, ejecutando nuestro programa de la siguiente forma:

```
java TSP solucion < ficheroprueba.txt > salida.dat
```

Esto hará que la solución se guarde en el fichero de texto `salida.dat` en lugar de aparecer en pantalla.

Descarga de la plataforma docente, el fichero `script`. Ejecuta `gnuplot`, e introduce la orden:

```
gnuplot> load "script"
```

A continuación introduce el comando `quit` para salir de `gnuplot`.

```
gnuplot> quit
```

En el directorio actual se habrá creado el fichero `grafica.png` que contendrá una imagen con la ruta calculada por tu programa, similar a las de la figura 1.

6 Mejoras propuestas no es obligatoria

Las tareas incluidas en esta sección no son obligatorias. Se utilizarán para subir nota. Como mejoras a la práctica, se propone implementar otros esquemas de solución.

6.1 Soluciones que usan estrategias de inserción

En primer lugar se propone el uso de *estrategias de inserción*. La idea es comenzar con un recorrido parcial, que incluya algunas de las ciudades, y luego extender este recorrido insertando las ciudades restantes con algún criterio a definir. Para poder implementar este tipo de estrategia, deben definirse tres elementos:

1. Cómo se construye el recorrido inicial parcial
2. Cuál es la siguiente ciudad a insertar
3. Dónde se inserta

El recorrido inicial se puede construir a partir de las 3 ciudades que formen el triángulo más grande: por ejemplo, eligiendo la ciudad que está más a la izquierda, la que está más a la derecha, y la que está más arriba.

Una vez seleccionada la ciudad a insertar, ésta será ubicada en el punto que provoque el mínimo incremento en la longitud de la ruta.

Para decidir cuál es la siguiente ciudad a insertar, debemos implementar dos estrategias (acopladas con los dos procedimientos anteriores):

- *Inserción más económica* (Algoritmo 2): entre todas las ciudades no visitadas, elijo aquella que provoque el mínimo incremento de la longitud. Éste es un enfoque puramente greedy. O sea, el funcionamiento sería el siguiente:
  - Se parte de una ciudad inicial v_0 cualquiera (que podría escogerse al azar).
  - Mientras queden ciudades por recorrer, se busca la ciudad no visitada v_i que insertada en la posición j de la ruta actual, produzca el menor incremento de distancia.
- *Inserción del más lejano* (Algoritmo 3): 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.

En caso de implementar estos algoritmos de solución, se añadirá un parámetro adicional (`algo`) al programa `main()` para elegir el algoritmo deseado. Por ejemplo, para elegir el algoritmo 2, usariamos:

```
java TSP algo 2 solucion < ficheroprueba.txt
```

6.2 Soluciones con algoritmos de Monte Carlo

Otras soluciones a este problema, son las que usan números aleatorios para encontrar las soluciones. Se propone implementar un algoritmo (Algoritmo 4) que genere K permutaciones (rutas) aleatorias del conjunto de ciudades y devuelva el mínimo valor encontrado. Asuma $K = 10000$.

Otro algoritmo (Algoritmo 5) que probablemente obtendrá mejores soluciones, consiste en generar una ruta aleatoria `rutaActual`, que luego se intentará mejorar haciendo cambios en la ruta actual. Un posible cambio podría consistir en poner una ciudad número j justo después de una ciudad i en la ruta actual. El cambio se haría si se mejora el coste de la ruta.

En el siguiente código se supone que la clase `Ruta` dispone del método `int siguiente(int nciudad)` el cual dada la ciudad número `nciudad` (no confundir este número con el orden de la ciudad en la ruta), nos

devuelve el número de la siguiente ciudad de acuerdo con esta ruta. También suponemos que la variable `ruta` contiene la ruta actual, que estamos intentando mejorar. Un esquema del algoritmo podría ser como el siguiente:

```
public void optimizar() {
  optimizacionLocal(nIteraciones);
}
private void optimizacionLocal(int niter) {
  int count = 0; // número de iteraciones seguidas
                 // sin encontrar una mejor solución
  double lmin = ruta.coste();
  do {
    optimizacionLocal();
    double len = ruta.coste();
    if (len < lmin) { //si esta ruta es más corta
       lmin = len;
       count = 0;
    } else {
       count++;
    }
  } while (count < niter);
}
private void optimizacionLocal() {
  ruta.random(randomGenerator); // generar ruta aleatoria
  while (mejorarRuta());
}
private boolean mejorarRuta(){
  for (i = 0; i < numeroCiudades; i++) {
    j=ruta.siguiente(ruta.siguiente(i))
    while (j != i) {  // Bucle que busca primera ciudad j que haya
              // después de i, para insertarla justo después de i
      if (coste de la ruta se reduce al insertar la ciudad
                          número j justo después de la i) {
         ruta.moverCiudadjDespuesCiudadi(j, i);
         return true;
      }
      j = ruta.siguiente(j);
    }
  }
  return false;
}
```

Como puede verse, el método público `optimizar()` usa un número entero `nInteraciones` que es el número de iteraciones seguidas que se permiten sin que se produzca una mejora. Podríamos fijarlo por ejemplo al valor 30, o bien introducirlo de algún modo, como parámetro de `main()`.

		Algo 1			Algo 2			Algo 3		
	Óptimo	min	% opt	mili	min	% opt	mili	min	% opt	mili
tsp 1	100	110	10%	130	100	0%	320	115	15%	420
...	...	...	...	...	...	...	...	...	...	...
tsp n	1000	1100	10%	150	1300	30%	430	1500	50%	200
Promedio			8.5%			7.3%			20%	

Table 1: Ejemplo de Tabla dónde se compara el rendimiento de cada algoritmo en cada instancia del problema. Ver descripción completa en el texto

6.3 Análisis comparativo

En esta tarea, evaluaremos el rendimiento de las estrategias que haya 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.

Al final, obtendremos una tabla como la tabla 1, donde cada fila resume los resultados para cada instancia del problema. La columna *Óptimo* indica el valor óptimo del recorrido para cada instancia y luego para cada algoritmo, se anotan tres valores: la longitud de la ruta obtenida, el porcentaje de exceso sobre el valor óptimo y el tiempo de ejecución en milisegundos que ha empleado el programa para calcular la solución. Para el cálculo del tiempo de ejecución en milisegundos podemos modificar el `main()` de la siguiente forma:

```
public static void main(String args[]){
  long time_start, time_end;
  time_start = System.currentTimeMillis();
  ... // Calculo de la solución del TSP
  time_end = System.currentTimeMillis();
  System.out.println("El tiempo de ejecución es " +
        ( time_end - time_start ) +" milisegundos");
}
```

La última fila de la tabla registra el promedio de los porcentajes y nos da una idea global respecto al rendimiento de cada algoritmo en el conjunto de prueba utilizado. Es decir, aquel algoritmo que obtenga el menor promedio, será el mejor para este conjunto de instancias.

Como última tarea, se debe construir un *ranking* que indique en cuantas instancias, la estrategia iesima fue la mejor.

7 Material a entregar

La entrega de la práctica debe constar de:

- **Análisis del problema**:

  El análisis del problema debe entregarse en un documento en formato pdf cuyo nombre debe ser `informe.pdf`. El documento irá

identificado en una portada por medio del nombre del alumno. Contendrá un índice, al principio del documento, y, en su caso, una bibliografía al final.

Este informe debe incluir:

  - **Diagrama de clases en UML**. Para hacer el diagrama de clases, puede emplearse una herramienta como **dia** (`http://projects.gnome.org/dia/`, o bien **umbrello** (http://uml.sourceforge.net/).
  - Justificación de la solución, convenientemente explicada, mediante la explicación de qué representa cada clase usada y detallando una por una todas las relaciones existentes entre las clases.
  - En caso de que se hayan implementado algunas de las mejoras, se explicarán con más detalle en una sección independiente del informe llamada *Mejoras implementadas*.

- **Manual de usuario del programa**, detallando todos los parámetros con los que se puede ejecutar el programa principal.

- **Proyecto netbeans** completo dónde esté incluido todo el código fuente, aunque no es necesario que esté compilado (no entregar los ficheros .jar, y .class).

  El código debe estar documentado usando `javadoc`. Para ello es recomendable leer el documento sobre `javadoc` disponible en `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`. Se requiere al menos añadir comentarios para:

  - Indicar qué representa cada clase.
  - Qué representa cada dato miembro.
  - Qué hace cada método (o constructor), qué valor devuelve, y qué representa cada uno de sus parámetros.

- **Documentación** `html` generada por `javadoc`

Todos estos ficheros (informe con el análisis del problema, proyecto netbeans con el código fuente y subdirectorio de javadoc) serán empaquetados en un archivo formato `tar.gz` que se llamará exactamente `practica4.tar.gz` y se entregará mediante la plataforma docente antes del fin del plazo establecido.