Curso 2019-2020
Estructura de Datos y Algoritmia
Práctica 3

Departamento de Lenguajes y Sistemas Informáticos

Objetivo

En esta práctica vamos a:

1. usar el `API Collections` de Java;
2. implementar en Java el tipo de datos grafo no dirigido y ponderado, y actualización de clases ya implementadas en la práctica 1 (en Java);
3. implementar una aplicación concreta con grafos y esquemas algorítmicos.

Plazo de entrega: desde el lunes 16 de diciembre hasta el **viernes 20 de diciembre**. Más información al final de este documento.

1. Clases

Hay que usar las siguientes clases implementadas en Java en la práctica 1:

- `Coordenadas`, donde hay que añadir el método:
  - `boolean equals(Object p)`: devuelve `true` si los dos enteros son iguales, `false` si no lo son;
- `InfoTur`
- `Localidad`
- `Coleccion`

Hay que implementar las siguientes nuevas clases:

- `Localidad2` *//clase que hereda de Localidad*
- `GrafLoc` *//grafo no dirigido y ponderado*
- `TNear` *//árbol que almacena las ciudades más cercanas a una distancia dada*

Podéis añadir las variables de instancia y métodos que consideréis necesarios en cualquiera de las clases anteriores para la realización de la práctica, justificando su necesidad.

Clase Localidad2

La clase `Localidad2` [1] que hay que implementar hereda de la clase `Localidad` y además contendrá:

- la siguiente variable de instancia (privada):
  - `vertice:` una variable de tipo entera para la nueva indexación
- y los siguientes métodos de instancia (públicos):
  - `Localidad2( Localidad l )`: se crea pasándole por parámetro una `Localidad` de la que coge todas sus características, e inicializa a -1 la variable `vertice`;
  - `int getVertice()`: método que devuelve el `vertice` asociado a la localidad;
  - `void setVertice( int i )`: método que no devuelve nada y recibe por parámetro un entero que asigna a la variable `vertice`;
  - `int compareTo( Localidad2 c )`: Este método establece una relación de orden entre dos objetos de tipo `Localidad2` utilizando como primer elemento de ordenación el nombre de la localidad y después el identificador de la localidad (`id`). El entero que se devuelve será:
    - -1 si el nombre de la localidad de la variable de instancia es anterior alfabéticamente al de la localidad del objeto pasado como parámetro;
    - 1 si el nombre de la localidad de la variable de instancia es posterior alfabéticamente al de la localidad del objeto pasado como parámetro;
    - si los nombres son iguales devolverá:
      - -1 si el resultado que devuelve el método `getId` para la localidad de la variable es menor al de la localidad del objeto pasado como parámetro;
      - 1 en caso contrario;
      - 0 si las dos localidades tienen el mismo identificador (es decir, estamos en la situación en la que ambos objetos tienen el mismo nombre de localidad y el mismo identificador).
  - `boolean equals( Object m )`: Este método devuelve `true` si ambas localidades son iguales, `false` si no lo son. Consideramos que dos localidades son iguales si tienen el mismo nombre y el mismo identificador [2].

[1] además esta clase tiene que implementar la interfaz `Comparable` de java ya que si no se hace se producirán errores de ejecución al realizar inserciones en el `TreeSet` o `TreeMap`. La cabecera quedará del tipo

```
public class Localidad2 extends Localidad implements Comparable<Localidad2>
```

[2] para que sea consistente con la definición de `compareTo`

Clase GrafLoc

La clase `GrafLoc` es un grafo no dirigido [3] y ponderado que cuando se use con las ciudades de un mapa almacenará en sus vértices el identificador de cada localidad y contiene:

- la variable de instancia siguiente (privada):
  - `ArrayList<ArrayList<Integer>> gr`: matriz dinámica que representa el peso de las aristas en una matriz de adyacencia
- y los siguientes métodos de instancia (públicos):
  - `GrafLoc( int n )`: crea la matriz de adyacencia [4] con todos sus valores al valor equivalente a ∞ `(Integer.MAX_VALUE)`;
  - `boolean esVacio()`: método que devuelve `true` si no existe ninguna conexión entre los vértices del grafo;
  - `boolean insertaArista( int o1, int o2, int p )`: método que crea (o actualiza) una arista no dirigida entre los vértices representados por los dos primeros enteros, y con peso el representado por el tercer entero. Si no se puede insertar la arista (o ya existe con el mismo peso p) se devuelve `false`, en caso contrario `true`;
  - `int recuperaArista( int o1, int o2 )`: método que devuelve el peso de la arista entre los vértices representados por los dos enteros pasados por parámetro. Si no se puede recuperar la arista se devuelve -1;
  - `boolean borraArista( int o1, int o2 )`: método que elimina del grafo, si existe, la arista que hay entre los dos vértices representados por los dos enteros pasados por parámetro. Si no se puede borrar la arista se devuelve `false`, en caso contrario `true`;
  - `boolean borraVertice( int o1 )`: método que elimina del grafo las conexiones del vértice representado por el entero pasado por parámetro. Si no se puede realizar la operación se devuelve `false` (incluso si ya no tiene ninguna conexión), en caso contrario `true`;
  - `int getVertices()`: método que devuelve el número de vértices del grafo;
  - `TreeSet<Localidad2> insertaLocalidades( Coleccion c )`: si el grafo no está vacío, lo vacía previamente. A continuación realiza las siguientes operaciones:

[3] por lo tanto su matriz de adyacencia será simétrica

[4] la matriz es cuadrada y la dimensión $n \times n$ (filas por columnas), siendo n el número de vértices pasado como parámetro si el valor es positivo, en caso contrario el número de vértices es 0

1. se recorren las localidades de la colección, creando sus correspondientes `Localidad2`, que se insertan en el árbol;
2. una vez insertadas se recorre el árbol, asignando el índice que ocupe en el árbol ordenado a su variable vértice [5]. Cada valor del nuevo índice se almacenará en la variable `vertice`;
3. se obtendrá la matriz de adyacencia en la que los vértices representan los índices anteriormente establecidos para las localidades, y las aristas se completan con la *distancia de Manhattan* calculada con las coordenadas de las localidades correspondientes a la fila y columna;
4. se devuelve el árbol obtenido;

London (21,10) -1	Roma (12,17) -1	Paris (2,3) -1	Munich (12,22) -1	Tabarca (1,20) -1	Bari (15,3) -1	London (4,9) -1

tras la ordenación, suponiendo un mapa de 100x100

Bari (15,3) 0	London (4,9) 1	London (21,10) 2	Munich (12,22) 3	Paris (2,3) 4	Roma (12,17) 5	Tabarca (1,20) 6

MATRIZ DE ADYACENCIA

	0	1	2	3	4	5	6
0	0	17	13	22	13	17	31
1	17	0	18	21	8	16	14
2	13	18	0	21	26	16	30
3	22	21	21	0	29	5	13
4	13	8	26	29	0	24	18
5	17	16	16	5	24	0	14
6	31	14	30	13	18	14	0

distancia entre Munich y Paris

[5] (primera ciudad un 0, segunda un 1, etc ...)

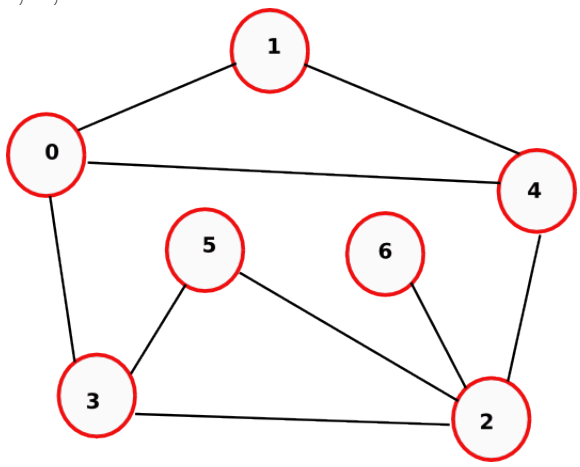
- `String toString()`: método que nos devuelve en un `String` el número de vértices, aristas y la matriz de adyacencia con el formato que se ve en el siguiente ejemplo.

  vertices: 7
  aristas: 21
  0 17 13 22 13 17 31
  17 0 18 21 8 16 14
  ...

- `void escribeDFS( int i )`: Es una generalización del recorrido en preorden (recorrido en profundidad). Hay que recorrer todos los vértices accesibles desde un vértice (entero pasado como parámetro). Cuando se termine, se continúa desde el siguiente que no haya sido visitado (de menor a mayor).

  En el grafo de la siguiente figura (que no se corresponde con la matriz de adyacencia anterior) empezando en 0 la salida de `escribeDFS` será:

  0, 1, 4, 2, 3, 5, 6



Clase TNear

La clase `TNear` que hay que implementar contendrá

- las siguientes variables de instancia básicas (privadas)

  - `TreeMap<Localidad2, TreeSet<Localidad2>> ti`; *//árbol de localidades (clave), cuyo valor asociado es un árbol con las localidades más cercanas a cada clave*

  - `int dn`; *//distancia máxima a la que pueden estar sus ciudades más cercanas*

- y los siguientes métodos de instancia básicos (públicos):

  - `TNear(int i)`: inicializa la variable `dn` a `i`, o a 0 en caso de que `i` sea negativo;
  - `boolean esVacio()`: nos indica si el árbol está vacío;
  - `void insertaLocalidad2(Localidad2 w)`: se inserta la localidad pasada como parámetro. Además, para la localidad, se tiene que almacenar un árbol asociado (un `TreeSet`) con las localidades que se encuentran a una distancia (*de Manhattan*) menor o igual que el valor `dn` a la localidad de referencia. En este proceso pueden ocurrir dos cosas diferentes:
    * si la localidad ya existe en el árbol no se hace nada;
    * si la localidad no estaba en el árbol, se añade un nuevo nodo al árbol con la localidad y hay que comprobar si hay que incluirla como una de las ciudades más cercanas al resto (según el valor `dn`), teniendo en cuenta su distancia, ya que el árbol puede no estar vacío inicialmente;
  - `void insertaLocalidades(Coleccion c)`: para cada localidad almacenada en la Colección pasada por parámetro se crea su correspondiente `Localidad2` y se inserta en el árbol con las condiciones especificadas en el método anterior.
  - `boolean borraLocalidad(String p)`: quita del árbol la localidad cuyo nombre sea igual al pasado como parámetro, devolviendo `true` si se ha podido realizar el borrado y `false` en caso contrario. Si la borra, también debe hacerlo en el árbol de localidades cercanas donde aparezca. Puede ocurrir que haya más de una localidad con el mismo nombre, por lo que debería borrarlas todas;
  - `TreeSet<Localidad2> getLocalidades(String s)`: devuelve el valor asociado (`TreeSet<Localidad2>`) de la primera localidad (clave) que se encuentre con ese nombre; si no existe ninguna localidad con ese nombre devuelve `null`;
  - `void setDn(int i)`: actualiza la variable interna `dn` al valor pasado como parámetro (si el parámetro es mayor que 0), modificando por tanto los `TreeSet` asociados a cada objeto de tipo `Localidad2` de forma que habrá que eliminar o añadir aquellos que no se ajusten al nuevo valor;
  - `TreeSet<Localidad2> getTop(int i)`: recibe un valor entero entre 0 y 3, y devuelve un `TreeSet<Localidad2>` con las localidades que tengan en su información turística un `top` con ese número de estrellas. Si se ha pasado un 0 se devolverán las localidades que no tengan `top`. Por defecto devuelve `null`;

– `String toString()`: escribe el árbol, empezando en la raíz, a la salida estándar, con el siguiente formato: nombre de la localidad, espacio en blanco, asterisco, espacio en blanco, nombre de las localidades asociadas separadas por espacio en blanco, guión, espacio en blanco menos la última que sólo lleva "\n". Ejemplo:

  localidad1 * locnear11 - locnear12 - locnear13 ...
  localidad2 * locnear21 - locnear22 ...
  localidad3 *
  ...
  localidadn * locnearn1 - locnearn2 - locnearn3 ...

2. Aplicación: algoritmo de Prim

El algoritmo de Prim es un algoritmo basado en una estrategia algorítmica voraz, que nos permite encontrar, sobre un grafo conexo y no dirigido, el **árbol de recubrimiento mínimo del grafo**, es decir, la manera menos costosa de conectar todos los vértices del grafo. Si el grafo está ponderado, el árbol de expansión mínima es el árbol cuyo peso (suma de los pesos de todas sus aristas) no es mayor al de ningún otro árbol de expansión.

Algoritmo de Prim

1. se marca un nodo cualquiera de salida;
2. se selecciona la arista de menor valor [6] conectado con el nodo marcado anteriormente, y se marca el otro nodo con el que conecta;
3. se repite el paso 2 (teniendo en cuenta todos los nodos marcados) siempre que la arista elegida enlace un nodo marcado y otro que no lo esté;
4. el proceso termina cuando tenemos todos los nodos del grafo marcados [7].

Se implementará una clase denominada `Prim` donde se ejecutará la aplicación, que consiste en encontrar el árbol de recubrimiento mínimo de un grafo, y que recibirá 2 parámetros:

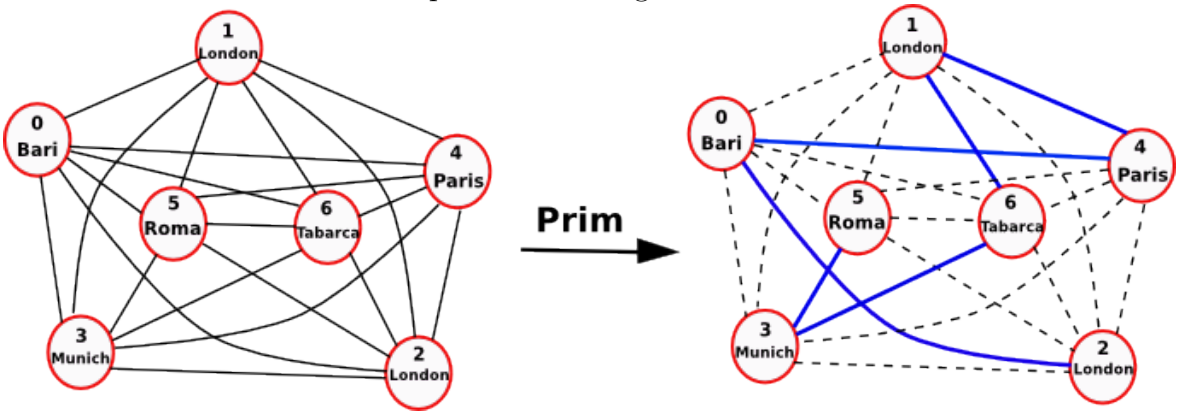
[6] en caso de que haya más de uno, se queda con la primera que encuentra
[7] en el tema de tipos no lineales de la asignatura tenéis un ejemplo

1. un fichero con la información del mapa y localidades del mapa;
2. un entero que indica el vértice en el que comenzamos la búsqueda;

En la clase se implementará un método `main` que:

- detectará los parámetros de la aplicación (un `String` y un entero mayor o igual a 0, en este orden);
- abrirá el fichero de texto con las localidades y lo almacenará todo utilizando los tipos de datos implementados;
- a partir de las estructuras de datos implementadas, construirá un grafo conectando todas las ciudades y utilizando como peso de cada arista la distancia entre las ciudades;
- sobre el grafo obtenido se aplicará el `algoritmo de Prim` y **mostrará por pantalla** todas las localidades adyacentes a cada vértice, uno por línea, según se detalla a continuación:
    1. una ciudad por línea, seguida por un guión, y a continuación sus ciudades adyacentes (acceso con una arista), ordenadas lexicográficamente y separadas por espacios en blanco
    2. la cadena "Distancia total", seguida de "=", y la suma del valor de las aristas de la solución terminando con un fin de línea

Suponiendo que el grafo de partida es el obtenido en la página 4, a continuación se muestra el resultado de la aplicación del algoritmo de Prim:



Salida para este ejemplo:
Bari-London Paris
London-Paris Tabarca

London-Bari
Munich-Roma Tabarca
Paris-Bari London
Roma-Munich
Tabarca-London Munich
Distancia total=66

3. Normas generales

Entrega de la práctica:

- **Lugar de entrega:** servidor de prácticas del DLSI, dirección `http://pracdlsi.dlsi.ua.es`;
- **Plazo de entrega:** desde el lunes 16 de diciembre hasta el **viernes 20 de diciembre a las 23:59 horas**;
- Se deben entregar las prácticas en **un fichero comprimido**, con todos los ficheros creados y ningún directorio de la siguiente manera:

  `tar cvfz practica3.tgz Coordenadas.java InfoTur.java Localidad2.java Localidad.java Coleccion.java TNear.java GrafLoc.java Prim.java`
- No se admitirán entregas de prácticas por otros medios que no sean a través del servidor de prácticas, siendo el usuario y contraseña para entregar prácticas el mismo que se utiliza en UACloud;
- La práctica se puede entregar varias veces, pero sólo se corregirá la última entregada;
- Los ficheros fuente deber estar adecuadamente documentados usando comentarios en el propio código, sin utilizar en ningún caso acentos ni caracteres especiales;

Sobre la evaluación en general:

- **MUY IMPORTANTE:** Los programas deben poder ser compilados sin errores ni warnings con el compilador de Java existente en la distribución de Linux de los laboratorios de prácticas; si la práctica no se puede compilar su calificación será 0. Se recomienda compilar y probar la práctica con el autocorrector antes de entregarla;
- **MUY IMPORTANTE:** Es imprescindible que se respeten estrictamente los formatos de salida indicados, ya que la corrección principal se realizará de forma automática;

- La práctica debe ser un trabajo original de la persona que entrega; en caso de detectarse indicios de copia de una o más entregas se suspenderá la práctica con un 0 a todos los implicados;
- La influencia de la nota de esta práctica sobre la nota final de la asignatura está publicada en la ficha oficial de la asignatura (apartado evaluación).

4. Probar la práctica

- En UACloud se publicará un corrector de la práctica con algunas pruebas (se recomienda realizar pruebas más exhaustivas).
- El corrector viene en un archivo comprimido llamado `correctorP3.tgz`. Para descomprimirlo se debe copiar este archivo donde queramos realizar las pruebas de nuestra práctica y ejecutar: `tar xfvz correctorP3.tgz`.

  De esta manera se extraerán de este archivo:

  - El fichero `corrige.sh`: *shell-script* que tenéis que ejecutar.
  - El directorio `practica3-prueba`: dentro de este directorio están los ficheros
    - `p01.java`: programa fuente en Java con un método main que realiza una serie de pruebas sobre la práctica.
    - `p01.txt`: fichero de texto con la salida correcta a las pruebas realizadas en `p01.*`.
- Una vez que tenemos esto, debemos copiar nuestros ficheros fuente (TODOS) al mismo directorio donde está el fichero `corrige.sh`.
- Sólo nos queda ejecutar el *shell-script*. Primero debemos darle permisos de ejecución. Para ello ejecutar:

  ```
  chmod +x corrige.sh
  corrige.sh
  ```

Apéndice: operaciones de TreeSet

Ejemplo de uso de iteradores para recorrer un `TreeSet`. En este ejemplo sólamente se visualiza el contenido de las claves (que son String).

```
Iterator<String> it;
TreeSet<String> ts = new TreeSet<String>();
ts.add("key"); ts.add("pencil"); ts.add("book"); ts.add("monkey");

it = ts.iterator();
while( it.hasNext() ){
  System.out.println( it.next() );
}
```

Otras operaciones:

- `void clear()`: borra todos los elementos del `TreeSet`
- `boolean contains(Object ob)`: devuelve verdadero si el objeto `ob` está en el `TreeSet`
- `boolean remove(Object ob)`: borra el elemento `ob`
- `int size()`: devuelve el tamaño del `TreeSet`
- `boolean add(E e)`: añade el elemento al `TreeSet`, si no está ya presente

Más información sobre el `TreeSet` en:

`https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html`