EDA (ETS de Ingeniería Informática). Curso 2020-2021

Practica 6: Una aplicación del algoritmo de Kruskal a la vida real: diseño del tendido eléctrico entre ciudades

Sesión 2: Obtención de un Árbol de Recubrimiento Mínimo "a la Kruskal"

Departamento de Sistemas Informáticos y Computación. Universitat Politècnica de València

1. Objetivos

Para cumplir con sus objetivos generales, al finalizar la segunda sesión de esta práctica el alumno deberá ser capaz de implementar el algoritmo de Kruskal de forma eficiente, i.e. reutilizando la Jerarquía Java `UFSet`.

2. Descripción del problema

Como se indicó en la primera sesión de la práctica, el conjunto de aristas que define un Árbol de Recubrimiento de un grafo No Dirigido y Conexo es solo una solución factible al problema de conectar con el menor coste posible sus `N` vértices mediante `N-1` aristas. La solución óptima a este problema pasa por encontrar un conjunto de `N-1` aristas del grafo tal que la suma de sus pesos sea mínima o, equivalentemente, que definan un Árbol de Recubrimiento Mínimo (ARM) del grafo. Aunque pueden existir varios conjuntos solución para un mismo grafo, el algoritmo de Kruskal garantiza obtener uno de ellos empleando una estrategia muy intuitiva:

> Procesar en orden creciente de pesos, una a una, las aristas del grafo (aristas factibles), incluyendo en el conjunto solución cada arista que no forme un ciclo con las ya incluidas en él -pues un Árbol de Recubrimiento es Acíclico por definición.

Obviamente, el proceso descrito puede terminar, bien cuando ya se han seleccionado las `N-1` aristas que definen un ARM del grafo, pues es Conexo, bien cuando ya no queda ninguna arista factible a procesar y aún no se han seleccionado `N-1` aristas, pues el grafo no es Conexo.

El siguiente esquema algorítmico resume el proceso de obtención "a la Kruskal" del conjunto de aristas que definen un ARM de un grafo No Dirigido, si es que existe. En él se introduce la notación que se empleará en el resto de la sección; a saber: `E` denota el conjunto de aristas del grafo y `aristasFactibles` el de las aristas del grafo aún por procesar; con el par `(v, w)` se nota la arista que conecta los vértices `v` y `w` del grafo, con `|E|` el número de sus aristas y con `E'` el conjunto de aristas que definen uno de sus ARM, `null` si el grafo no es Conexo; finalmente, se emplea el símbolo ∅ para denotar el conjunto vacío.

```
E' = ∅; cardinalE' = 0;
aristasFatibles = E;

mientras (cardinalE' < N - 1 && aristasFatibles != ∅):
    (v, w) = eliminarMin(aristasFatibles);
    Si ((v, w) NO forma ciclo con las aristas de E'):
        E' = E' UNION (v, w);
        cardinalE'++;
    FinSi
FinMientras

Si (cardinalE' == N - 1): solución = E'; Sino: solución = null; FinS
```

Nótese que una implementación eficiente del algoritmo presentado requiere que, en cada paso del proceso iterativo que describe, sea posible...

- Obtener y eliminar el mínimo de `aristasFactibles` (`eliminarMin`) de la forma más eficiente posible, i.e. en O(log`|E|`). Una forma de conseguirlo es representar el conjunto `aristasFactibles` como una Cola de Prioridad implementada mediante un Montículo Binario.
- Comprobar si la arista `(v, w)` forma ciclo con las aristas de `E'` de la forma más eficiente posible, i.e. en aproximadamente O(1). Dado que una arista NO forma ciclo si los vértices de sus extremos están en distintas componentes conexas, una forma de conseguirlo es representar las componentes conexas del grafo

definido por `E'` mediante un *UF-Set* `cc` de talla `N` e implementado "en Bosque" de forma eficiente. En efecto:

- Inicialmente, `E'` es un conjunto vacío de aristas que define un grafo de `N` vértices aislados; por tanto, el *UF-Set* `cc` está compuesto por `N` componentes conexas o `N` vértices aislados, cada uno en su propia componente conexa.
- En cada iteración, para comprobar si la arista `(v, w)` extraída de `aristasFactibles` forma ciclo con las de `E'` se tienen que determinar primero, vía operación `find` del *UF-Set*, las componentes conexas a las que pertenecen `v` y `w`:

  `int ccV = cc.find(v); int ccW = cc.find(w);`

  Hecho esto, si `v` y `w` están en distintas componentes, i.e. si `ccV != ccW`, se incluye la arista `(v, w)` en el conjunto solución `E'` y se actualizan las componentes conexas de `E'` vía operación `union` del *UF-Set* (`cc.union(ccV, ccW)`).
- Finalmente, si el grafo definido por las aristas de `E'` es Conexo, la iteración termina cuando la talla de `E'` es `N - 1`.

Usando las EDAs indicadas, la implementación del algoritmo de Kruskal obtiene un ARM de un grafo en O(`|E|` log`|E|`), exactamente el mismo coste asintótico que requiere procesar en orden creciente las 2`|E|` aristas que puede contener la Cola de Prioridad `aristasFactibles` en el Peor de los Casos. Para ser más concretos, el hecho de que el coste dominante sea el de las operaciones de la Cola de Prioridad y no el de las del *UF-Set* se debe exclusivamente al uso de una implementación "en Bosque" eficiente del *UF-Set*: en el Peor de los Casos, comprobar si 2`|E|` aristas forman ciclo supone realizar en tiempo constante `N - 1` operaciones `union` y 2`|E|` operaciones `find`, i.e. realizar O(`N` + `|E|`) operaciones.

3. Actividades

Antes de realizar las actividades que se proponen en esta sesión, el alumno debe actualizar varios paquetes de su proyecto *BlueJ eda* siguiendo los pasos que se indican a continuación. Todos los ficheros mencionados en ellos están disponibles en *PoliformaT* y se deben descargar en las carpetas correspondientes al paquete del mismo nombre de su proyecto *eda*.

- Descargar en la carpeta *modelos* el fichero `UFSet.java`.
- Descargar en la carpeta *jerarquicos* el fichero `ForestUFSet.class`, que contiene una implementación "en Bosque" eficiente de la interfaz `UFSet`.
- Descargar en la carpeta *grafos* el fichero `TestKruskal.class`.
- Abrir el proyecto *BlueJ eda* y compilar la clase `UFSet` de su paquete *librerias.estructurasDeDatos.modelos*. Hecho esto, salir de *BlueJ* seleccionando la opción *Salir* de la pestaña *Proyecto*.
- Invocar de nuevo a *BlueJ* y acceder al paquete *librerias.estructurasDeDatos.grafos* de su proyecto *eda*.

3.1. Actualizar la clase `Arista` e implementar el método `kruskal` de la clase `Grafo`

En esta actividad el alumno debe completar el código del método `kruskal` de la clase `Grafo` usando el algoritmo homónimo descrito en el apartado 2 de este boletín. En concreto, para tener en cuenta las consideraciones realizadas en dicho apartado sobre la implementación eficiente del algortimo de Kruskal, el alumno debe...

- Usar las clases `ColaPrioridad` y `MonticuloBinarioR0` de su proyecto *eda* para implementar la Cola de Prioridad `aristasFactibles`.
- Usar las clases `UF-Set` y `ForestUFSet` de su proyecto *eda* para implementar el *UF-Set* `cc`.
- Incluir las directivas `import` que aparecen comentadas en la clase `Grafo` para poder reutilizar los modelos e implementaciones Java de Cola de Prioridad y *UF-Set*.
- Modificar el código de la clase `Arista` para que implemente la interfaz `Comparable`, pues `aristasFactibles` es una Cola de Prioridad de `Aristas`.

3.2. Validar el código desarrollado en la práctica

Para comprobar la corrección del código implementado durante la sesión el alumno debe ejecutar el programa `TestKruskal`.