Estructuras de Datos y Algoritmos - EDA (ETS de Ingeniería Informática).
Curso 2023-2024

Práctica 6 - Parte 2. Una aplicación del algoritmo de Kruskal: diseño del tendido eléctrico entre ciudades

Obtención de un Árbol de Recubrimiento Mínimo (mediante algoritmo de Kruskal)

Departamento de Sistemas Informáticos y Computación. Universitat Politècnica de València

1. Objetivos

Para cumplir con sus objetivos generales, al finalizar la segunda parte 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 vio en la primera parte 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 sus `N` vértices mediante `N-1` aristas. La solución óptima es encontrar un conjunto de `N-1` aristas tal que la suma de sus pesos sea mínima (i.e., que definan un ARM del grafo). Aunque pueden existir varias solucines 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).*

Este proceso puede terminar cuando ya se han seleccionado las `N-1` aristas que definen un ARM del grafo (si es Conexo) o cuando ya no queda ninguna arista por procesar y aún no se han seleccionado `N-1` aristas (el grafo no es Conexo). Para presentar el esquema algorítmico, usaremos la siguiente notación:

- `E` denota el conjunto de aristas del grafo. `|E|` denota el número de sus aristas.
- `aristasFactibles` denota el conjunto de las aristas del grafo aún por procesar.
- Con el par `(v, w)` se denota la arista que conecta los vértices `v` y `w` del grafo.
- `E'` denota el conjunto de aristas que definen uno de sus ARM.
- El símbolo ∅ denota el conjunto vacío.

Teniendo en cuenta esta notación, el siguiente esquema algorítmico, de Kruskal, resume el proceso de obtención del conjunto de aristas que definen un ARM de un grafo No Dirigido:

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

mientras (cardinalE' < N - 1 && aristasFactibles != ∅):
   (v, w) = eliminarMin(aristasFactibles);
   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; FinSi
```

Una implementación eficiente del algoritmo requiere que, en cada paso del proceso iterativo, sea posible...

- Obtener y eliminar el mínimo de `aristasFactibles` (`eliminarMin`) de la forma más eficiente, 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, 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 `UFSet cc` de talla `N`. En efecto:
  - Inicialmente, `E'` es un conjunto vacío de aristas que define un grafo de `N` vértices aislados. Por tanto, el `UFSet cc` está compuesto por `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 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'`:

    ```
    cc.union(ccV, ccW);
    ```
  - Finalmente, si el grafo 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. El hecho de que el coste dominante sea el de las operaciones de `ColaPrioridad`, y no el de las de `UFSet`, se debe al uso de una implementación "en Bosque" eficiente de `UFSet`: 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 propuestas en esta parte, se actualizarán varios paquetes del proyecto *BlueJ eda* siguiendo los pasos dados a continuación. Todos los ficheros mencionados están disponibles en *PoliformaT* y se deben descargar en las carpetas correspondientes de los paquetes que se indican.

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

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

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

- Usar las clases `ColaPrioridad` y `MonticuloBinario` para implementar la Cola de Prioridad `aristasFactibles`.
- Usar las clases `UFSet` y `ForestUFSet` para implementar el `UFSet cc`.
- Incluir las directivas `import`, que aparecen comentadas en la clase `Grafo`, para poder reutilizar los modelos e implementaciones de `ColaPrioridad` y `UFSet`.
- Modificar el código de la clase `Arista` para que implemente la interfaz `Comparable`, pues `aristasFactibles` es una Cola de Prioridad de objetos `Arista`.

3.2. Validar el código desarrollado

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