24 | Gráfica con listas de adyacencia

Meta

Que el alumno implemente una *gráfica* haciendo uso de *listas de adyacencia* y los principales algoritmos para trabajar con ellas.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Aplicar las características de las estructuras de datos vistas anteriormente para implementar una estructura más compleja.
- Hacer uso de estructuras auxiliares para la implementación de algoritmos sobre gráficas.

Antecedentes

Estructura

La implementación de una gráfica utilizando listas de adyacencia elabora en la metodología utilizada para representar a los árboles binarios haciendo uso de vértices y referencias, pero con el ingrediente adicional de aristas que pueden tener un peso asociado. Para esta práctica trabajarás con gráficas dirigidas pesadas. Cada vértice almacenará en su interior un dato, que también le servirá como etiqueta para distinguirlo de los otros vértices. La Figura 24.1 muestra las clases que usarás, con sus atributos y métodos. En esta ocasión harás uso de tres clases internas: `Vértice`, `Arista` e `IteradorAmplitud`; el ser internas les permitirá acceder al tipo genérico `T`.

La clase `Arista` contiene referencias al vértice donde incia `v1` y al vértice hacia el que apunta `v2`. Mientras que cada vértice guardará en su lista `vecinos` a aquellas aristas que

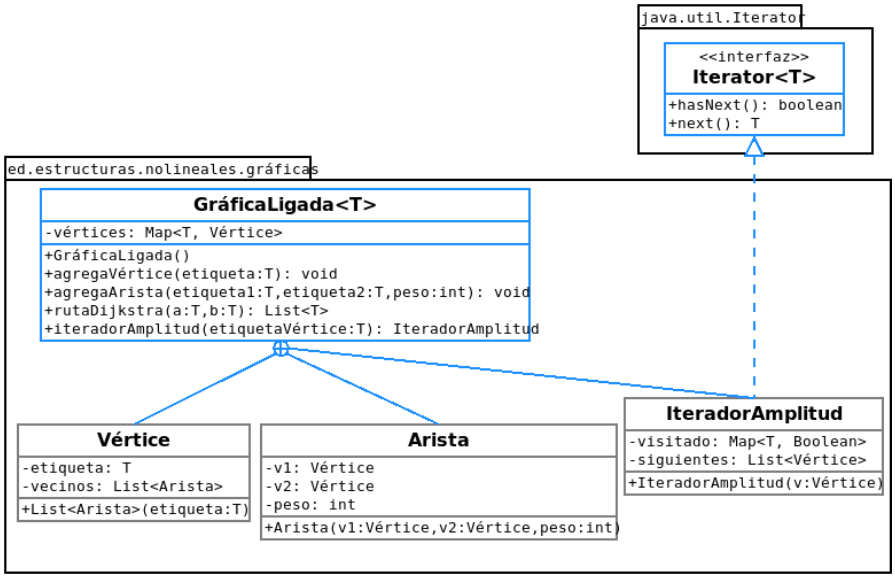

Figura 24.1 La gráfica ligada contiene tres clases internas.

salgan de él. Sería posible tener también una lista de aristas que apunten al vértice, pero no la necesitarás en esta práctica.

La clase `GráficaLigada<T>` tendrá como atributo una tabla de dispersión que indexe a los vértices según su etiqueta. Entonces, dada la etiqueta de un vértice podrás acceder al vértice y, a partir de él, a las aristas que emergen de él.

Observa que en esta ocasión `GráficaLigada<T>` no implementa `Iterable<T>`. Esto es porque cualquier recorrido requiere que se indique el vértice inicial, elegir uno al azar sería demasiado arbitrario y el método `iterator()` no recibe argumentos. Pero sí programarás un método `iteradorAmplitud()`, que reciba la etiqueta del vértice inicial como parámetro, y devuelva un iterador que recorra la gráfica en amplitud a partir de él.

Cuando un usuario haga uso de tu clase, su código se verá como el siguiente:

```
1  public class UsoGráfica {
2    public static void main(String[] args) {
3      GráficaLigada<Character> g = new GráficaLigada<>();
4      g.agregaVértice('A');
5      g.agregaVértice('B');
6      g.agregaArista('A', 'B', 10);
7      Iterator<Character> it = g.iteradorAmplitud('A');
8      while(it.hasNext()) {
9        System.out.println("Vértice␣" + it.next() + "␣visitado.");
10     }
11   }
12 }
```

Observa que todos los atributos de las clases internas son accesibles desde la clase

externa, aunque su acceso sea privado. Maneja estos atributos con cuidado.

Iteradores

Programa sólo el iterador en amplitud. Utiliza una cola como atributo del iterador para guardar los vértices a visitar. En cada llamada a `next()` devuelve la etiqueta del nodo extraído de la cola. El recorrido termina cuando la cola esté vacía.

Desarrollo

1. Crea la clase `GráficaLigada<T>` en el paquete

   `ed.estructuras.nolineales.gráficas`

2. Programa las clases internas `Vértice` y `Arista` según el diagrama UML. Cuida que la etiqueta del vértice no puede ser `null`, ni los vértices conectados por la arista.

3. Agrega la tabla de dispersión a `GráficaLigada<T>` y un constructor que inicialice una gráfica vacía.

4. Agrega los métodos:

   - `public void agregaVértice(T etiqueta)`. Crea un vértice con la etiqueta indicada y lo almacena en la tabla de dispersión.
   - `public void agregaArista(T etiqueta1, T etiqueta2, int peso)`. Crea la arista conectando a los vértices representados por las etiquetas y la almacena en la lista del vértice inicial.
   - Añade el iterador en amplitud.
   - Programa el método `List<T> rutaDijkstra(T a, T b)` que reciba las etiquetas de dos vértices en la gráfica y devuelva la lista de etiquetas de vértices que correspondan a un camino de peso mínimo entre los vértices con etiquetas `a` y `v`.

     Observa que las estructuras auxiliares necesarias para implementar este algoritmo serán sólo variables locales a este método, no necesitas añadir atributos a la clase.
   - Prueba y documenta tu código.

Preguntas

1. Dado el diseño de esta clase es posible tener una gráfica donde un vértice esté desconectado de todos los demás. Si quisieras impedirlo ¿Cómo lo harías?

2. ¿Qué necesitarías para implementar el algoritmo de Prim en otra función?